LISP STYLE & DESIGN

MOLLY M. MILLER / ERIC BENSON

LISP STYLE & DESIGN

MOLLY M. MILLER / ERIC BENSON

LISP STYLE & DESIGN

mm™
Digital Press

Copyright © 1990 by Digital Equipment Corporation.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

9 8 7 6 5 4 3 2 1
Order number EY-C199E-DP

Printed in the United States of America.

Cover and text design: David Ford
Editing and coordination: Alice Cheyer
Composition: Paul Anagnostopoulos (using Z^TgX with PostScript output)
Production: Editorial Inc.

The following are mentioned in this book: CLX, © 1987, Texas Instruments, Inc.; GNU Emacs, © 1988, Richard Stallman and the Free Software Foundation, Inc.; Lucid Common Lisp, a trademark of Lucid, Inc.; MACSYMA, a trademark of Symbolics, Inc.; PostScript, a trademark of Adobe Systems Inc.; UNIX, a trademark of AT&T Bell Laboratories; T^X, a trademark of the American Mathematical Society; the Digital logo, a trademark of Digital Equipment Corporation.

Library of Congress Cataloging-in-Publication Data

Miller, Molly M., 1953-
Lisp style & design / Molly M. Miller, Eric Benson,
p. cm.
Includes bibliographical references.
ISBN 1-55558-044-0
1. LISP (Computer program language) I. Benson, Eric, 1955—
II. Title
QA76.73.L23M54 1990 005.13*3-dc20 89^9296

TO SUSAN BENSON AND IN MEMORY OF WILL MILLER, JR

CONTENTS

Preface ix

1 INTRODUCTION 1

1.1 Style and Communication 1

1.2 Style and Lisp 2

1.3 Our Approach to Style 5

1.4 Typographical Conventions 8

2 GETTING STARTED 9

2.1 Understanding a Programming Problem 9

2.2 Subdividing the Programming Problem 15

2.3 Writing Experimental Code 20

3 CHOOSING THE BEST CONSTRUCTS 31

3.1 Steps Toward Good Code 31

3.2 Lisp Data Structures 32

3.3 Appropriateness of Lisp Constructs 37

3.4 Programming Idioms 60

4 PUTTING THE PIECES TOGETHER 65

4.1 Using Program Tools Well 65

4.2 Abstraction 65

4.3 Intermodule Interfaces 88

4.4 Fine-Tuning a Program 89

5 COMMENTING AND ORGANIZING THE FILES 99

5.1 Introduction 99

5.2 Self-Commenting Code 100

5.3 Useful Commentary 102

5.4 Code File Organization 105

6 DEBUGGING 114

6.1 The Style of Debugging 114

6.2 Lisp Debugging Tools 115

6.3 Debugging Methodology 133

6.4 Program Maintenance 137

7 EFFICIENCY 138

7.1 Introduction 138

7.2 Programming Efficiently 138

7.3 Declarations and Compilation 142

7.4 Generalizing Program Code 148

Appendix A: The Personal Planner 151

Appendix B: A Sample Session 198

Appendix C: Suggested Exercises 205

Bibliography 206

Index 209

PREFACE

Lisp Style & Design is built on the premise that programming style distinguishes a good programmer from an average one. Though style is an important consideration with any programming language, it is a more interesting issue with a symbolic language like Lisp, which allows a wide range of natural expression. Good style aids program readability, maintenance, and debugging. Good style habits lead to improved organization skills and thus are useful problem-solving tools.

In this book, we show how to design, write, document, and debug well-organized programs that use the complete expressive power of Lisp effectively. We illustrate our ideas by tracing the development of a Lisp program, the *Personal Planner*, and by drawing on code from other programs. Our discussion is supplemented by a sample interaction with the *Personal Planner*, suggested exercises, and a bibliography.

This book is not a Lisp primer; we assume our readers have written Lisp programs before. Readers with a basic knowledge of Lisp can benefit from our recommendations for improving their program-writing style, which have evolved from our own practice and from conversations with other experienced Lisp programmers.

Molly M. Miller wrote the text of this book, and Eric Benson wrote the *Personal Planner* as well as many of the other programming examples. Jussi Ketonen gave us the idea for this book, and many of the programmers at Lucid, Inc., contributed information toward this effort. Richard P. Gabriel, Wade Hennessey, Ken Olum, Robert Poor, and Leonard Zubkoff reviewed early drafts of the manuscript and made many useful suggestions for improving its organization and exposition. We are also grateful to reviewers Paul Anagnostopoulos, Jack Beidler, and Henry Ledgard. George Horesta and Chase Duffy at Digital Press helped us to bring our ideas into print. Paul Anagnostopoulos braved the terrors of TEX for us, and David Ford made our work look interesting. Alice Cheyer carefully reviewed our writing and judiciously improved it.

Special thanks are due to Robert Poor, who tirelessly answered questions and listened to complaints about the book, and to Susan Benson, who has put up with many late nights of Lisp hacking for ten years.

Chapter 1

INTRODUCTION

1.1 STYLE AND COMMUNICATION

There are many types of style guides, such as those that suggest techniques for writing prose well, those that recommend procedures for publishing, and those, like this one, that suggest methods for programming more effectively. All these types of style guides have a common theme—to improve the process of communication.

A computer language, above all else, is a language, a means of communicating knowledge and ideas through a specialized vocabulary. Programs are the medium for communicating through computer languages. Programs represent the transfer of ideas from one programmer to another and from a programmer to a computer. Some programmers focus on the latter exchange while neglecting the former. However, it is equally, or perhaps more, useful to improve the understanding that programmers can gain from reading a program, since programmers must use, correct, maintain, and revise programs. As Abelson and Sussman have said, "Programs must be written for people to read, and only incidentally for machines to execute" (p. xv). [1]

As with any language, computer languages can be used well or misused. In this book, we attempt to discriminate between these extremes. Any information in this book should be considered a recommendation, not an inflexible doctrine.

Regardless of the computer language, good style makes programs easier to understand, to debug, and eventually to maintain; that is, stylistically good programs make more sense. This clarity is the result of two aspects of style:

- Good style implies good organization, which shortens the path to the solution of a programming problem.

1. Full references are given in the Bibliography at the end of the book.

- Good style is incompatible with ambiguous, vague, or otherwise obscure information.

A well-written program often has the added benefit of executing more efficiently.

1.2 STYLE AND LISP

Lisp is a symbolic programming language, that is, a programming language whose vocabulary consists of symbols. While some languages, like Fortran, were designed to manipulate numbers, Lisp was designed to work on symbols that can have a very wide range of uses and interpretations. Though Lisp has often been associated with artificial intelligence applications, it has proved to be a general-purpose language. For example, a popular text editor from the Free Software Foundation, GNU Emacs, is written in Lisp. Two widely acclaimed computer algebra systems, REDUCE and MACSYMA, are also Lisp applications.[2] And this book shows that Lisp is even well suited for a more mundane personal tool, such as a daily appointment calendar.

Common Lisp is a dialect of Lisp designed by a committee of prominent Lisp programmers to serve as a portable, consistent, and expressive Lisp standard. It incorporates many features of former Lisp dialects as well as a common core of indispensable language constructs for the Lisp programmer. In this book, any references to Lisp imply Common Lisp.

Lisp is an easily extensible language, since you can treat Lisp functions in the same way as you treat Lisp data; that is, functions are first-class objects. All user-written functions and macros are assimilated into the Lisp environment and receive the same type of treatment as predefined Common Lisp constructs. The value of a variable can be any Lisp object, such as a number, character, symbol, list, array, structure, or function. Lisp provides anonymous functions that can be passed as arguments to other functions. As a consequence, Lisp allows a variety of useful data and procedural abstractions, and every Lisp programmer invents an application-specific language, a language that incorporates Common Lisp while extending it to suit particular needs.

Lisp objects are typed objects. In other words, you do not have to declare a Lisp variable to be of a certain type before using it. The type is associated with the value of the variable in a given context, not with its symbol name. A single symbol can represent many different objects, depending on the context. For example, you could define a Lisp array and name it `my-array`. You could specify that the array would hold ten different elements and that every element would be of a different Lisp type. In this case, the expression `(aref my-array 3)` would return an object of a different type from the expression `(aref my-array 7)`, even though both objects are represented by the symbol `my-array`.

2. Refer to Hearn and to Moses for information about these applications.

Consider the following examples, which present code for creating a normalized random list. The first example shows how such an operation could be captured in a C program:

```
/* The random() function is part of the Unix library. */
long random();

/* The float_or_long union is used to allow an array to contain
   either float or long elements. */
typedef union {long long_part; float float_part} float_or_long;

/* generate_normalized_random_list takes a result vector of
   length n and fills it with normalized random numbers.  The sum
   of all the elements of result is 0.0. */
void
generate_normalized_random_list(result, n, max_value)
     float_or_long result[];
     int n;
     long max_value;
{
  float average;
  long total;
  int i;

  total = 0;
  /* First fill result with random integers less than max_value */
  for (i = 0; i < n; i++)
    total += result[i].long_part = random() % max_value;
  /* Take the floating point average */
  average = (float)total / (float)n;
  /* Normalize the elements of result and convert to float */
  for (i = 0; i < n; i++)
    result[i].float_part = result[i].long_part - average;
}

main()
{
  int i, n;
  long max_value;
  float_or_long *result;
```

```
    for (;;)
      {
        printf("Size: ");
        scanf("%d", &n);
        printf("Maximum value: ");
        scanf("%D", &max_value);
        result = (float_or_long *)malloc(n * sizeof(float_or_long));
        generate_normalized_random_list(result, n, max_value);
        printf("Result is (");
        if (n > 0)
          printf("%f", result[0].float_part);
        for (i = 1; i < n; i++)
          printf(" %f", result[i].float_part);
        printf(")\n");
        free(result);
      }
}
```

Note that every variable used in these procedure definitions is explicitly typed before it is used and that the specified type of any variable cannot be changed during the execution of the procedure. The type is associated with the name of the variable, not with its value. We must even specify the type of user input being read. Any iteration control variables must be explicitly initialized, and storage for pointers must be explicitly allocated. Note also the rigid format of these definitions and the amount of punctuation they require in order to be correctly parsed.

In contrast, consider the operation as recoded in Lisp:

```
;;; GENERATE-NORMALIZED-RANDOM-LIST returns a sequence of normalized
;;; random numbers.  N is the number of elements of the sequence.
(defun generate-normalized-random-list (n max-value)
  (loop repeat n
        for value = (random max-value)
        sum value into total
        collect value into denormalized-list
        finally
        (return (loop for element in denormalized-list
                      with average = (/ total n)
                      collect (- element average)))))
```

```
(defun test ()
  (loop
    (format t "~&Size: ")
    (let ((n (read)))
      (format t "~&Maximum value: ")
      (let* ((max-value (read))
             (result (generate-normalized-random-list n max-value))
             (*print-array* t))
        (format t "Result is ~S~%" result)))))
```

The types of the variables `n`, `max-value`, `value`, `total`, `denormalized-list`, and `average` are never declared in the Lisp definitions. The user input is read into an arbitrary symbol; that symbol can hold any type of data. All memory management is handled automatically; nowhere is storage explicitly allocated. The expressions are easy to read, since the syntax is less constrained by artificial punctuation.

Most of the punctuation in Lisp programs consists of opening and closing parentheses. Lisp does not require many punctuation markers because the Lisp reader understands the allowable syntax for Lisp constructs and thus can interpret the meaning of symbols by their position and context. This syntactic simplicity allows a wide variety of expression in Lisp, but it also puts a burden on Lisp programmers to write code whose purpose is clear and well defined.

Common Lisp provides dynamic as well as lexical scoping. If a variable has no valid binding in the lexical context of an expression, it is assumed that there is a dynamic, or global, variable of that name whose value can be used in the given situation. Lisp programmers must account for the differences in scoping and must work to advertise the correct interpretation of a variable in any given context.

The wide range of flexibility available in Lisp makes it a very powerful programming language. Combined with programming environments that most Common Lisp vendors provide with the language, Lisp is a tool for successful program development. Style is an essential ingredient of any well-developed program, but it takes on additional significance in Lisp programs because of Lisp's flexibility. Since Lisp programmers have more ways to express solutions to their programming problems, there are more stylistic issues to consider, and there is a greater likelihood of divergences in style. Therefore, in this book we catalog our own style recommendations for Lisp, which arise from our own practice as well as from conversations with other experienced Lisp programmers.

1.3 OUR APPROACH TO STYLE

We consider style not an attribute but a process. Our approach to style comprises efficient algorithms, good organization, appropriate abstractions, well-constructed function defi-

nitions, useful commentary, and effective debugging. Our rationale can be summed up in the following way:

- If the individual elements of a program are well written, but the overall organization of the program is poor, the program will not be stylistically satisfying.
- If a program is well organized, but the individual function definitions are hard to understand, the program will not be stylistically satisfying.
- If the program is well organized and well written, but the algorithm chosen does not adequately address the problem, the program will not be stylistically satisfying.

The style of a well-written program does not need to call attention to itself; rather, it serves as a transparent aid to both the programmer's and the reader's understanding of the solution to a problem. By thinking of style as a process rather than as an end in itself, we can achieve a balance of subtlety and effectiveness in style.

In this book, we explore the process of style for Lisp programs—the book is not a Lisp primer.[3] We assume our readers have written Lisp programs before. Note that we sometimes smooth over the details of Common Lisp to simplify our discussion; readers should not take us so literally that they miss the points we are trying to make. Our descriptions of Lisp are meant to illustrate style concepts, and we often refer to other books on Lisp that give more complete information. Two of these books deserve special mention here: *Common Lisp: The Language*, by Guy L. Steele Jr., is the definitive Common Lisp specification, and *The Common Lisp Object System*, by Jo A. L. Haskell and Molly M. Miller, describes how to use constructs in the Common Lisp Object System, CLOS.

We discuss the process of style through the following chapters.

- In Chapter 2, Getting Started, we present suggestions for organizing an approach to a program. A well-thought-out approach is generally the most important ingredient of good programming style, and there is no substitute for it. To illustrate our points, we show the abstract description and implementation plan for a sample program called the *Personal Planner*. We also present some initial code for that program to preview some of the concepts described in detail in later chapters.
- In Chapter 3, Choosing the Best Constructs, we compare and contrast Lisp data structures and a variety of control constructs. We also describe situations when certain constructs are more effective or appropriate than others. Finally, we review some Common Lisp idioms and present some points to consider about idiomatic programming.

3. We recommend *A Programmer's Guide to Common Lisp*, by Deborah Tatar, as an excellent textbook for those readers who are new to Lisp.

- In Chapter 4, Putting the Pieces Together, we discuss abstraction techniques available to Lisp programmers. We bring up some issues to consider when constructing interfaces so that the connections between Lisp modules maintain the distinctions between the modules. We also explore the iterative process of fine-tuning a program through an example from the *Personal Planner*.
- In Chapter 5, Commenting and Organizing the Files, we suggest ways of documenting your code and of organizing the finished pieces of a program.
- In Chapter 6, Debugging, we describe the debugging tools built into Common Lisp and present techniques for effectively using those tools to correct and maintain Lisp programs.
- In Chapter 7, Efficiency, we discuss techniques for improving both program and programmer efficiency.
- In Appendix A, The Personal Planner, we show the program listing for the *Personal Planner*. We have used code from this program as the basis for many of the examples in this book.
- In Appendix B, A Sample Session, we interact with the sample program.
- In Appendix C, Suggested Exercises, we list a few enhancements to the sample program that could be done by the reader.

The *Personal Planner* is a Common Lisp program for managing personal information. Its scope is limited, since we wanted to fit all the code into a reasonable number of pages; thus, it lacks some features that more complete applications might have, such as a graphic user interface or a long-term storage mechanism. These restrictions also limit the number of style issues that it addresses. Other code examples that are not related to the *Personal Planner* demonstrate some of the concepts in the text.

The book charts the growth of the *Personal Planner* from the original idea for the program to the execution of the completed code. We do not describe every modification or rewrite—that would quickly become tedious for everyone—but we have tried to capture the program's evolution honestly, since the process is as important as the result. Thus, you might notice that portions of the program that are described in early chapters become obsolete by the end of the book.

All examples in this book have been run under Lucid Common Lisp; output might vary across other implementations.

The *Personal Planner* may be ordered from Lucid, Inc., Attn: Personal Planner, 707 Laurel Street, Menlo Park, CA 94025 (please include your phone number), or by electronic mail to `personal-planner@lucid.com`. Specify required format and medium:

diskette, reel-to-reel tape, or cartridge tape. There is a nominal fee to cover media and duplication costs.

1.4 TYPOGRAPHICAL CONVENTIONS

User-defined Lisp forms and terminal input and output are set in a monospaced font, as are Lisp symbols being described in the text. Required arguments to Lisp functions and new terms are set in an italic font.

Chapter 2

GETTING STARTED

2.1 UNDERSTANDING A PROGRAMMING PROBLEM

A veteran Lisp hacker once said that he attacks every Lisp programming problem by just typing `(defun ...`. This approach may be commonly used, but it is not the shortest route to a well-written program.

The best first step in solving a programming problem is to describe the problem to yourself. You cannot expect an adequate solution unless you understand the problem in your own terms. Ask yourself some basic questions:

- What needs to be done? Describe the general goals that the program must achieve.
- What are the big pieces of the problem? Consider the steps required to reach the overall solution of the task. How do the pieces fit together?
- What is the scope of the problem? Try to identify issues that concern you as well as those you can ignore.
- What resources are available to you as the programmer? Common operations are often captured in library routines. Often you can find examples of programs that perform similar tasks or that provide a certain part of your solution. In many cases, you can discuss the reasoning behind the example code with the programmer who wrote it. Avoid doing work that has already been done. However, you must be aware that someone else's model for a solution may not be compatible with yours.
- What kind of input is supplied and who supplies it? Certain programs require user interaction, while others receive input from other programs.
- How does the program demonstrate that the desired task is complete? The program might succeed quietly by calling other programs or by properly initializing a device. It could also return formatted results, including sophisticated charts and tables.

- Who will use the program? A novice user might require a simple and friendly interface, while a sophisticated user might expect more features and some access to the program itself.
- What constraints limit the approach to the problem? Often programming projects must conform to strict deadlines, or the resulting application must be compatible with existing programs or must run within a limited amount of physical memory. The environment of the target system might also lack certain resources.

Conceptualizing the problem might give you some ideas about its solution and how that solution might best be expressed. Thus, by answering questions about the problem, you can begin to develop a program plan. The initial plan does not need to be elaborate, just a simple written description of the problem and some possible solutions. You can expand the plan as you flesh out the skeleton of the program by writing bits and pieces of code. These fragments will spark more questions, which will lead to different portions of the solution.

2.1.1 Initial Description of Programmatic Interactions

To illustrate this first step, describing the problem, let's consider the program that illustrates many of the principles in this book, the *Personal Planner*. We want a program that can chronicle someone's day-to-day business affairs—that is, schedule appointments, keep track of names and addresses, and organize information in different ways, such as by activity, person, or affiliation. A user of this program should be able to enter, modify, or retrieve information easily at any time.

Instead of collecting and connecting scattered bits of information in its database, the program can construct information *objects*, each with a unique set of characteristics. For example, every person that the *Personal Planner* knows about can be stored as a person object. A person object contains such information as the person's job title, address, and phone number. It might include information about the person's employer in the form of an organization object, which has its own independent components. The user can access any object as a whole as well as any component of an object.

An object type hierarchy outlines the relationships objects have with one another. This arrangement allows storing information within a context so that the user can retrieve useful chunks of information instead of patches of data that form some vague mosaic. The Common Lisp Object System (CLOS) is well suited to this approach, since it provides the natural abstractions to accommodate these needs. Section 2.3.1 provides more detail about this aspect of our program planning.

Now that we have some ideas about the basic tasks the *Personal Planner* can perform, we need a way to communicate with the program. The best approach requires fancy graphics with pop-up menus and mouse capabilities. However, Common Lisp does not provide a graphics standard yet, and we also want to limit the size of the program so that it can be printed in its entirety in this book. Thus, we leave it as an exercise to the reader to devise a graphical interface to this program (see Appendix C).

Instead, two simple interfaces are required: a *command interface* and a *programmatic interface*. A user issues commands to the *Personal Planner* through the command interface. The command interface reads simple expressions, such as "Describe Joy Rorex," "Schedule dentist on May 1 at 2:00," "Show appointments for April 1." The programmatic interface consists of a set of Lisp functions that establish the object world and provide access to it from the command interface. The programmatic interface manipulates Lisp objects in response to the user's commands and enters or returns the appropriate information.

Users supply information about persons, organizations, addresses, phone numbers, and meetings, and the *Personal Planner* stores that information in a global database.

The program prompts the user for additions, modifications, and deletions to the database and provides means for manipulating stored information. For example, if the user wants to schedule a meeting, the program prompts for pertinent information, such as the date, time, and location of the meeting, those scheduled to attend it, and the reason it was called. After the user enters the information, the program checks the database for scheduling conflicts. If no conflicts exist, the program reserves that time in the database; otherwise, it informs the user of the problem. The user can then ask the program to suggest an alternative time for the meeting based on the information available. After the meeting has occurred, the user can enter some notes about the business that took place for later review.

Once the user enters information into the *Personal Planner*, any piece of that information can be retrieved and related to any other piece of information that has already been cataloged, such as

- Meetings scheduled for a particular date
- Meetings scheduled with a particular person
- Persons connected with a given organization
- Persons located at a specified address
- Date of last phone conversation or meeting with a person
- Persons whose phone numbers begin with a certain area code
- Persons with a given first name that work for a particular organization

- Notes on a given person, organization, or meeting

In effect, the user gets more information out of the program than was put into it. The ability to store objects as composite collections of data allows the program a wide range of manipulative ability.

2.1.2 The Implementation Plan

From this initial description, we can write up an implementation plan.

The *Personal Planner* is a personal information manager; it provides a computer analog of a personal appointment calendar, address book, and diary. It is a single-user program to be written in portable Common Lisp.

The function `personal-planner` is the top-level function of the program; it enters a command parser when called. Input will be supplied by the user in a simple command language.

The program provides an easy-to-understand terminal-based user interface as well as a program interface. The program interface facilitates support for alternative user interfaces, especially window-based interfaces. Maintainers of the program can easily add new types of objects without modifying the user interface.

Because the *Personal Planner* is a single-user program, its database is handled in a simple manner. We do not try to address problems such as multiple access to the same data.

The information stored in the database is highly structured, not simple text. Things in the real world, such as people, organizations, and meetings, are represented by objects that are stored in a table so that they can be accessed by name. The operations performed by the *Personal Planner* include finding objects, displaying the contents of objects, creating new objects, removing objects, and modifying existing objects.

The major pieces of the program—maintaining the database and providing a good user interface—require the following steps:

1. Creating an infrastructure for storing objects
2. Defining the major object types
3. Creating the user interface
    a. Defining an input language
    b. Making descriptive output for each of the major object types
    c. Implementing creation, deletion, and modification commands

The steps are interrelated in several ways. We must coordinate the infrastructure for storing objects with the definition of the objects and with the user interface for creating, deleting, and modifying objects. The object definitions provide the information needed for displaying the contents of the objects.

Since this program serves as an example of good programming style, we are willing to sacrifice speed in the interest of program clarity if necessary. For example, we can use CLOS classes to represent object types, even though the program might run faster if we chose a different representation.

The program must be written in portable Common Lisp and must be short and simple enough to be described in this book. Barring such restrictions, we would use a window-system interface such as CLX. [1]

Now that we have a simple understanding of the problem and some directions for supplying the pieces of the solution, we can write a little code to test our ideas and to expand our knowledge of the problem. As we learn more about the problem, we can commit our ideas to a program design. Using experimentation as part of the design process challenges established practices of programming from a strict and complete specification. The interactive nature of Lisp and its supporting environment allow us this flexibility in design.

2.1.3 A Closer Look at the Program

In response to the plan, we must first consider in more detail the types of objects that the program manipulates. We must be able to represent the subjects of the program—people and organizations—as well as their actions and interactions. To begin, we specify that the *Personal Planner* has three main types of objects: people, organizations, and events.

We can explore these objects by writing some simple definitions of them. For the purposes of this exercise, we cast these definitions as `defclass` forms, since it is likely that we will use either classes or structures during prototyping. Let's start with the objects we understand best—people, organizations, and addresses:

```
(defclass person ()
  (name
   employed-by
   position
   address
   phone-number))
```

1. See "CLX—Common Lisp Language X Interface."

```
(defclass organization ()
  (name
   address
   phone-number))

(defclass address ()
  (street-number
   street-name
   city
   zip-code))
```

Now, what should an event look like? We need to be able to identify an event and to specify when it takes place, where it will happen, and how long it will last. What types of events will most people need to schedule? We can begin with these event definitions:

```
(defclass event ()
  (starting-time
   duration
   location))

(defclass meeting (event)
  (attendees
   agenda
   description))

(defclass phone-call (event)
  (to-person
   originated
   regarding))

(defclass diary-entry (event)
  (description))
```

Simply writing out these definitions gives us a fuller understanding of the types of information available to the *Personal Planner*. From this understanding, we can explore the options for using that information.

2.1.4 Design Iteration

The design process is a continuous one. The steps that are described in the following sections help round out the design strategy formulated in this most important conceptual phase. Every time you look further into the problem, your picture of the task becomes more concrete. Sometimes this new understanding requires you to modify earlier notions; at other times you can just add a new layer to an existing part of the solution. In some

cases, entirely new pieces appear and other pieces become obsolete. A good design is strengthened by a critical review and the evolution that proceeds from it. It is important not to become so completely wedded to one approach that you are blinded to an improved one. In the long run, you will find that it is less work to fine-tune a good design than to work around a poor one.

2.2 SUBDIVIDING THE PROGRAMMING PROBLEM

Once you have seen the problem as a whole, it is important to look at the individual pieces so that the shape of each piece can be examined and made to fit into the whole puzzle. Take your programming problem apart.

- What are the most obvious parts of the problem? The steps that lead to the solution of the general task are likely divisions.
- What are the most obvious subparts of those parts? Iterate over the larger steps until you understand what they comprise. Think about incremental solutions to the problem.
- Do certain parts depend on others? The order in which your program develops might be constrained by dependence among the pieces. Can you break any of this dependence? Removing constraints allows for greater program modularity.
- Which parts are essential and which are embellishments? Get the important operations working properly before you add the bells and whistles. If you arrange the pieces by priority, you might find some areas that could be left out if time becomes an issue.
- How can you work around the pieces that you don't quite understand? Are there effective ways for dealing with partial knowledge? You might find that working through one piece will broaden your understanding of a different piece. If possible, it is often a good idea to begin with those pieces you understand best or that you can conquer most easily.

This exploratory surgery provides not only useful directions for writing code but also a clearer and more complete picture of the programming problem. If you understand the elements that make up the program, you are less likely to encounter unwelcome surprises or insurmountable obstacles midway through the project. You are more likely to end up with a modular design, one that combines many discrete tasks to solve a larger problem.

2.2.1 The First Piece of the Puzzle

Now that we have a working description of the data handled by the *Personal Planner*, we can tackle the issue of storing the database. (Our modular approach relieves us of any

constraints to solve the pieces of the problem in some sequential order.) We need a way to record new objects and to save changes made to objects.

A simple approach to storing the database involves a file of entries that is read in every time the program starts up. The program converts the entries into a table of objects that can be manipulated by Lisp. Upon exit, the program writes the modified entries into the file.

Thus, we end up with two representations for every object:

- The file representation
- The Lisp object in memory

However, we only have to deal with one of these representations at a time.

This solution could become unsatisfactory if the *Personal Planner* had to handle large numbers of objects at once. As the number of objects in the database increases, the amount of Lisp memory required to store them also grows; thus, we have no control over the size of our application. Also, the length of time it takes to load in, to restore, and to save the program increases as the database expands.

Though this might not be the best final solution, we can use this simple approach while developing the other pieces of the program, since we do not need to worry about speed of access at first. As the program matures, we might discover important aspects of the storage problem that are not clear at this conceptual stage. Once we understand the issues, and if time and resources permit, we can implement a more sophisticated storage system.

One possibility for expanding the simple storage mechanism involves storing most of the objects in the file while the application is running and keeping only a small cache of objects in Lisp memory. Thus, the program would handle the two types of object representation concurrently. When the user asked a question about an object, the program would first search the cache for the object; if the object were not available, the search would proceed to the file. If the object were found in the file, a corresponding Lisp object would be created and added to the set of objects in main memory. If no such object existed, the user could create one, which would be stored in the cache.

This approach requires that the file be ordered in some way so that searching for an object would be efficient. When an object in the cache is modified, the file must be updated immediately to reflect the change.[2] This virtual storage solution would work well for the *Personal Planner*, since it is intended for the personal use of a single user. This

2. Lisp programmers have used the Metaobject Protocol, a proposed addition to the Common Lisp Object System, to create a class of persistent objects; see Paepcke. Such a feature would allow us to develop a write-through cache with minimum changes to the completed version of the *Personal Planner* in this book.

enhancement is left as an exercise to the reader (see Appendix C). To support multiple users, this approach would have to be upgraded to prevent simultaneous access to both representations of an object by different users.

2.2.2 The Remaining Pieces of the Program

In addition to a working storage facility for the *Personal Planner*, we need a way to query the database. We must be able to locate an object, to create an object, and to change an existing object. A *programmatic interface* is a set of functions that serves as a link between the user of a program and the database that stores program information. The following partial list of operations is our first attempt at creating a programmatic interface for the *Personal Planner*. This list includes functions that create objects, predicates that identify whether an object is of a particular type, and accessors that retrieve components of each object. There are functions to locate events and entities and to perform some specifiable operation on them. In addition, there are functions for scheduling and canceling events and for recording notes about them.

```
;;; Entity--state.
;;; Constructor, predicate, and accessors.
(MAKE-STATE name abbreviation)
(STATE-P object)
(STATE-NAME state)
(STATE-ABBREVIATION state)

;;; Entity--city.
;;; Constructor, predicate, and accessors.
(MAKE-CITY name state)
(CITY-P object)
(CITY-NAME city)
(CITY-STATE city)

;;; Entity--address.
;;; Constructor, predicate, and accessors.
(MAKE-ADDRESS street city zip)
(ADDRESS-P object)
(ADDRESS-STREET address)
(ADDRESS-CITY address)
(ADDRESS-ZIP address)
```

```
;;; Entity--phone number.
;;; Constructor, predicate, and accessors.
(MAKE-PHONE-NUMBER area-code local
                   &optional extension)
(PHONE-NUMBER-P object)
(PHONE-NUMBER-AREA-CODE phone-number)
(PHONE-NUMBER-LOCAL phone-number)
(PHONE-NUMBER-EXTENSION phone-number)

;;; Entity--organization.
;;; Constructor, predicate, and accessors.
(MAKE-ORGANIZATION name address phone-number)
(ORGANIZATION-P object)
(ORGANIZATION-NAME organization)
(ORGANIZATION-ADDRESS organization)
(ORGANIZATION-PHONE-NUMBER organization)

;;; Entity--person.
;;; Constructor, predicate, and accessors.
(MAKE-PERSON name address phone-number &optional organization title)
(PERSON-P object)
(PERSON-NAME person)
(PERSON-ADDRESS person)
(PERSON-PHONE-NUMBER person)
(PERSON-ORGANIZATION person)
(PERSON-TITLE person)

;;; Event--meeting.
;;; Constructor, predicate, and accessors.
(MAKE-MEETING time location topic attendees)
(MEETING-P object)
(MEETING-TIME meeting)
(MEETING-LOCATION meeting)
(MEETING-TOPIC meeting)
(MEETING-ATTENDEES meeting)

;;; Searching mechanism for entities.
(FIND-ENTITIES name)

;;; Perform operation on an entity.
(MAP-ENTITIES name function)

;;; Perform operation on all entities.
(MAP-ALL-ENTITIES function)
```

```
;;; Enter new item in datebook.
(SCHEDULE-EVENT event time location)

;;; Change existing entry in event database.
(RESCHEDULE-EVENT event old-time new-time)

;;; Remove entry from event database.
(CANCEL-EVENT event)

;;; Searching mechanism for events.
(FIND-EVENTS starting-time ending-time)

;;; Perform operation on an event.
(MAP-EVENTS starting-time ending-time function)

;;; Add a diary entry about an event.
(MAKE-DIARY-ENTRY event)
```

Writing this outline helps us to identify all the pieces of the programmatic interface. Thinking about a function brings to mind the arguments it needs; thinking about the arguments brings to mind other functions that would use them.

Next we turn our attention to the user interface. How do these pieces look to the user? Identifying what the user wants to know about these objects helps us to examine the way in which information might be obtained and stored. We can fabricate output for a hypothetical function named `display` to help us understand the user-visible parts of objects:

```
> (DISPLAY phone-number)
(415) 555-8400

> (DISPLAY address)
707 Laurel Street
Menlo Park, CA 95022

> (DISPLAY organization)
Lucid, Inc.
707 Laurel Street
Menlo Park, CA 95022
(415) 555-8400
```

```
> (DISPLAY person)
Eric Benson
Principal Scientist
Lucid, Inc.
707 Laurel Street
Menlo Park, CA 95022
(415) 555-8400

> (DISPLAY meeting)
Sales Meeting
4 April 1989 1:00pm
Marketing Conference Room
Attendees:
Eric Benson
Molly Miller
```

Finally, let's sketch out how the user interacts with the *Personal Planner* through a simple menu-style interface:

```
A(dd entry), F(ind entry), C(hange entry), D(elete entry), Q(uit)
```

Any one of these options has several parts:

```
Add entry:
P(erson), O(rganization), M(eeting), E(xpense), ..., O(ther)
```

When a user selects an option, the program prompts for data:

```
Person:
Name: <typein>
Title: <typein>
Organization: <typein>
Address: <typein>
Phone-number: <typein>
```

With these plans for structuring the program, we can soon begin coding in a logical direction and in an organized fashion.

2.3 WRITING EXPERIMENTAL CODE

A Lisp program is a machine-runnable manifestation of a design. You can demonstrate your ideas early in throwaway code before committing yourself to the overall design of the solution. What seems like a reasonable approach while you are mulling over the problem in the shower often looks less robust when you are staring at the debugger prompt. Writing

simple print functions provides a good way to view your data structures in all of their beauty and flaws.

The interactive environment of Lisp makes it convenient for you to load in a file or two and experiment with the code. If you find problems in your code, you can either modify it from the debugger or edit the file and dynamically reload it. For example, by coding interactions for the early class definitions presented in Section 2.1.3, we found that some slots were not very useful and others were missing. We simply revised our definitions, loaded in the new definitions, and then reevaluated the interactions. Soon we had a working object base to build upon.

In the following sections, we show some early code for the *Personal Planner* and describe the reasoning that led to it. Some of these definitions might disappear in the final version of the code, but they are important as steps in reaching the finished program. They provide us with information about what works in our design and what does not.

We show this code to demonstrate coding and style concepts that we describe in later chapters. For now, we concentrate on the thought processes that occur while writing experimental code.

2.3.1 Personal Planner Data Structures

We cannot get very far in our coding without making some assumptions about data structures. See Section 3.2 for a description of Lisp data structures and the issues surrounding them. For our purposes here, however, we simply mention the reasoning behind our choice of data structures for prototyping.

For the *Personal Planner*, we want data structures that can model objects of different composition and complexity but that still allow for uniform treatment of all objects. Structures might work for our objects, since they allow us to distinguish between objects by defining their characteristics as slots that have predefined accessor and constructor functions. With this solution, one object also can inherit some properties of another structure.

What type of objects are listed in our notes for the functional interface? Addresses include information about cities and states. Persons use information about organizations. Clearly, inheritance would be a useful feature for sharing characteristics.

In addition to providing for uniform treatment of complex objects and the ability to share characteristics, we must accommodate many types of events over time; that is, our program is demand driven. As the user needs to schedule different types of events, we need to add new types of objects to the program. For example, we might want to add an account object to represent a bank account. We could then add the capability to handle transactions—an exchange of funds between accounts.

Both inheritance and program extensibility are easily modeled with CLOS. Classes let us tailor the structure of an object to fit its characteristics without duplicating information already defined by an object. We can manipulate new class objects by adding methods to existing generic function interfaces. In addition to these legitimate concerns, we simply want to try out this new capability of Common Lisp.

To see how CLOS would work for the *Personal Planner*, let's write definitions for a few classes, beginning with entities:

```
;;; Entities are locations, people, organizations, i.e., everything
;;; except events.  Every entity has a name and a list of nicknames.
(defclass entity ()
  ((name :type string
         :accessor entity-name)
   (nicknames :type list
              :initform '()
              :accessor entity-nicknames)))
```

The essence of an entity is that it has a name, which is a string. It can also have one or more nicknames. Any object that inherits from this class inherits the ability to have a name and nicknames, as well as accessor functions for retrieving them.

Now let's define a couple of types of entities:

```
(defclass person (entity)
  ((phone-number :type phone-number)
   (organization :type organization)
   (address :type address)
   (title :type string)))

(defclass organization (entity)
  ((address :type address)))
```

Our primary reasons for using the `:type` slot option are to describe features of the class and to point out other new types we need as classes—phone numbers and addresses. However, it might also serve as advice to the compiler to optimize our code. Since both persons and organizations require information about phone numbers and addresses, we can simplify our definitions by taking advantage of inheritance. We simply add a new entity class as an abstract layer to entity objects.

```
(defclass locatable-entity (entity)
  ((address :type address)
   (phone-number :type phone-number)))
```

We can now revise the definitions for persons and organizations to take advantage of this new class:

```
(defclass person (locatable-entity)
  ((organization :type organization)
   (title :type string)))

(defclass organization (locatable-entity)
  ())
```

This change leaves us with a sparse definition of organizations. We might expand this definition later, when we have a better idea of the types of information it must cover.

These explorations into entities reveal more definitions that we need to write, but they also show that classes work well for entities. How would they work for events? To find out, let's try an event class definition:

```
(defclass event ()
  ((starting-time :type integer)
   (location :type location)
   (duration :type integer :initform 0)))
```

This definition seems simple enough and covers our needs; an event happens at a certain time in a particular place and ends sometime later.

To bolster our confidence in this object-oriented approach, let's take another step and consider the location class required by events. Since a location can be any place, such as a city, a state, or a street address, it is best to divide a location into components by defining a class from which other location types can inherit:

```
;;; Every location has a time zone.  Of course, a state may have
;;; more than one time zone.  Any one of the time zones in the state
;;; will suffice, since events are rarely scheduled for an entire
;;; state.
(defclass location (entity)
  ((time-zone :accessor location-time-zone)))

;;; States also include Canadian provinces and foreign countries.
(defclass state (location)
  ((abbreviation :type string)))

;;; Cities with the same name can exist in different states.
(defclass city (location)
  ((city-name :type string)
   (state :type state)))
```

These definitions confirm our choice of CLOS as a workable approach to handling the needs of the *Personal Planner*. Our class definitions are simple, and we can easily extend them through inheritance.

2.3.1.1 THE ENTITY DATABASE

Because the entity superclass contains all classes of named objects, we clearly must be capable of storing many of these objects. These objects will be accessed frequently and quickly, since they are the subjects of all of the program's actions. Each entity is identified by name or nickname only; it has no numeric identifier. Thus, a hash table seems the best first choice for the entity database:

```
(defvar *entity-table* (make-hash-table :test #'equalp))
```

We use an `equalp` hash table because it is going to hold strings that we compare in a case-insensitive manner. (Unfortunately, the function `make-hash-table` does not support a `#'string-equal` test.) The operators for this database are built upon the Common Lisp hashing functions and thus are quite simple. For example, the following functions respectively add, find, and remove entities from the database:

```
;;; Install an entity by name.  Since any entity can have multiple
;;; names, the name stored with each entity is its preferred name.
;;; Objects are compared for identity with an EQ test.
(defun add-entity (name entity)
  (pushnew entity (gethash name *entity-table*) :test #'eq))

;;; Return a list of all entities with the given name.
(defun find-entities (name)
  (gethash name *entity-table*))

;;; Remove the entity by name.
(defun remove-entity (name entity)
  (let* ((entries (gethash name *entity-table*))
         (remaining-entries (delete entity entries :test #'eq)))
    (cond ((null remaining-entries)
           (remhash name *entity-table*))
          ((eq entries remaining-entries))
          (t (setf (gethash name *entity-table*)
                   remaining-entries)))))
```

These definitions do not depend on any particular representation for entities. They would work just as well for structures as for classes.

2.3.1.2 THE EVENT DATABASE

We want to store events by time, since time is the most important piece of information for ordering events and for preventing the collision of events. We also need quick access to event objects. A chronological list of events will not work, since the search time might get quite long as the number of events increases. We also need to locate events at a specific time as well as over a range of time, such as every event that happens tomorrow. Since we cannot access more than one hash table element at a time, we can use a binary tree structure instead. The binary tree structure has four slots—the left and right branches, the time the event occurred, and the events stored at a node:

```
(defstruct (binary-tree)
  (left-branch '())
  (time (get-universal-time))
  (entries '())
  (right-branch '()))
```

Searching for events on this binary tree is quick because of the simple way in which the events are sorted—those on left branches occurred earlier than the event at their node, and those on right branches occurred later. Once a branch has been passed up, all its subnodes can be ignored because they fall in the wrong time period. Any number of event items can be returned by accessors for the slots because the slot definitions are simply lists of objects. To find objects within a range of time, we simply traverse the tree between the times specified as the endpoints of the range.

2.3.2 Program Utilities

A useful by-product of writing experimental code is that it provides fodder for writing other parts of the program. For example, a small but pervasive piece of the *Personal Planner* is time, which is necessary for scheduling appointments and for maintaining a diary of events. Before other aspects of the design can make sense, it is important to understand how time is represented and manipulated. Since time operates independently of the operations that use it, it also serves as a nice vehicle for easing deeper into the programming project.

We can write a simple date-parsing utility by taking advantage of Lisp's Universal Time format. This utility can be used generally by any other function.

```
(defun parse-date (string time-p)
  "Read a date from the string.  If time-p is true, read a time as
   well.  Return an integer in Universal Time format.  The date and
   time must be in the form: 3 April 1988 11:29 pm.  Three-letter
   abbreviations are allowed for the month.  If a year is given
   between 50 and 99, 1900 will be added to it.  If a year is
   given between 0 and 49, 2000 will be added."
  (multiple-value-bind (day-of-month string-position)
      (parse-integer string :junk-allowed t)
    (check-type day-of-month (integer 1 31))
    (multiple-value-bind (month-name string-position)
        (read-from-string string t nil :start string-position)
      (check-type month-name symbol)
      (multiple-value-bind (year string-position)
          (parse-integer string :start string-position
                                :junk-allowed t)
        (check-type year (integer 0 2100))
        (let ((month-number (position month-name
                                      month-names :test #'member)))
          (unless (and (integerp month-number)
                       (< 0 month-number 13))
            (error "~S is not known to be the name of a month"
                   month-name))
          (let ((adjusted-year (if (< year 100)
                                   (if (< year 50)
                                       (+ year 2000)
                                       (+ year 1900))
                                   year)))
            (multiple-value-bind (hour minute)
                (if (not time-p)
                    (values 0 0)
                    (multiple-value-bind (hour string-position)
                        (parse-integer string
                                       :start string-position
                                       :junk-allowed t)
                      (check-type hour (integer 0 24))
                      ;; Skip over :
                      (multiple-value-bind (minute string-position)
                          (parse-integer string
                                         :start
                                         (1+ string-position)
                                         :junk-allowed t)
                        (check-type minute (integer 0 60))
```

```
                ;; Look for AM or PM.  If neither, use
                ;; 24-hour clock.
                (let ((am-or-pm (read-from-string
                                  string nil string
                                  :start string-position)))
                  (values (if (eq am-or-pm 'pm)
                              (+ hour 12)
                              hour)
                          minute)))))
          (encode-universal-time 0 minute hour
                                 day-of-month month-number
                                 adjusted-year)))))))
```

This function converts a date written by the user into Universal Time format, an integer representation that can be easily manipulated by other functions. As it scans each substring of the date, it simultaneously performs the conversion and binds the variable `string-position`, which marks the position of the substring that was scanned. After checking that the result of the conversion is reasonable, it scans the next substring of the date, beginning at `string-position`. It converts the only non-numeric portion of the date string, the name of the month, into an integer from 1 to 12. Once all the substrings have been converted to integers, the function has all the arguments it needs to call `encode-universal-time` to finish the operation.

Now to locate objects stored by time, we need only search for a given integer value instead of performing tedious string comparisons. The user's model of time remains unchanged.

To make this utility complete, we need a way to convert back from the internal representation to the user's representation of dates. In writing the reciprocal operation, we can draw upon the knowledge of Universal Time format that we gained in writing `parse-date`. Any flaws in our understanding will become apparent if we cannot easily convert the integer back into a date string.

```
(defun write-date-and-time (universal-time
                            stream
                            &key exclude-day exclude-time timezone)
  (multiple-value-bind (sec min hrs date month year
                        day daylight-p time-zone)
      (if timezone
          (decode-universal-time universal-time timezone)
          (decode-universal-time universal-time))
```

```
(format stream
  ;; Print the day, followed by one space, if EXCLUDE-DAY is
  ;; false.  Print the date as a decimal string, padded on
  ;; the left with spaces; the month as a three-character
  ;; string; and the year as a two-character string, padded
  ;; on the left with zeros.  If EXCLUDE-TIME is false, print
  ;; the hours, minutes, and seconds as two-digit, zero-padded
  ;; fields, separated with colons.  Finally, print any time
  ;; zone as a three-character uppercase string.
"~:[~*~;~@(~A~) ~]~2D-~@(~A~)-~2,'0D~@[~* ~2,'0D:~2,'0D:~2,
 '0D~@[ ~A~]~]"
(not exclude-day)
(day-name day t)
date
(month-name month t)
(mod year 100)
(not exclude-time)
hrs
min
sec
(if (<= 5 time-zone 8)
    (svref (if daylight-p
               '#("EDT" "CDT" "MDT" "PDT")
               '#("EST" "CST" "MST" "PST"))
           (- time-zone 5))
    nil))))
```

The function `decode-universal-time` does the first step for us—it breaks up the Universal Time code into its constituent parts, which are bound to variables by `multiple-value-bind`. Then, by exercising the functionality of `format` and by using vector operations, we complete the translation.

After defining the constants used by the date functions, we can test this interface by loading these functions into a running Lisp:

```
> (setq birthdate (parse-date "11 Nov 1953 12:38" t))
1699735080
> (write-date-and-time birthdate t)
Wed 11-Nov-53 12:38:00 PST
NIL
```

Our definition of an event includes not only the starting time but also the duration of the event. Thus, we need a way to capture a time interval. We can build this definition upon our previous ventures through time:

```
(defun write-time-interval (seconds stream short-p)
  ;; Write seconds to stream in the form:
  ;; 32 days, 4 hours, 5 minutes, 23 seconds.
  (multiple-value-bind (days seconds)
      (truncate seconds seconds-per-day)
    (multiple-value-bind (hours seconds)
        (truncate seconds seconds-per-hour)
      (multiple-value-bind (minutes seconds)
          (truncate seconds seconds-per-minute)
        (if short-p
            ;; If short-p is true, use this form:
            ;; 32 04:05:23.
            (format stream "~@[~D ~]~2,'0D:~2,'0D:~2,'0D"
                    (if (> days 0) days nil) hours minutes seconds)
            (let ((comma-needed nil))
              (flet ((print-component (amount name)
                       (when (> amount 0)
                         (when comma-needed
                           (write-string ", " stream))
                         (format stream "~D ~A" amount name)
                         (unless (eql amount 1)
                           (write-char #\s stream))
                         (setq comma-needed t))))
                (print-component days "day")
                (print-component hours "hour")
                (print-component minutes "minute")
                (print-component seconds "second")
                (unless comma-needed
                  ;; Nothing else has been printed,
                  ;; so it is an empty interval.
                  (write-string "0 seconds" stream)))))))))
  seconds)
```

This function shows the number of days, down to the second, that a given number of seconds spans. As you can see, we combine techniques from `parse-date` and `write-date-and-time` by using the Common Lisp `flet` construct for writing local function definitions. The local function prints all the components of the interval and their identifiers whenever the short interval format is not specified. It even determines when to use the singular and plural forms of the identifiers. We need more power than `format` to print these components, since `format` does not save the state of its arguments. Here is how this function looks when run:

```
> (write-time-interval 12345678 t nil)
142 days, 21 hours, 21 minutes, 18 seconds
12345678
> (write-time-interval 12345678 t t)
142 21:21:18
12345678
```

The function `parse-date` encodes the starting time when the event is entered in the database as the key by which the event is stored. When the user asks about a particular event, the print method for the event object calls the functions `write-date-and-time` and `write-time-interval` to provide time information about the event:

```
(defmethod print-object ((object event) stream)
  (with-slots (starting-time location duration) object
    (format stream "#<~:(~A~) at " (class-name (class-of object)))
    (write-date-and-time starting-time stream
                         :timezone
                         (location-time-zone starting-location))
    (when (> duration 0)
      (write-string " for " stream)
      (write-time-interval duration stream nil))
    (write-char #\> stream)))
```

The user sees the information in the following format:

#<*name-of-event* `at` *date-and-time* `for` *duration*>

2.3.3 The Next Step: Fleshing out the Program

We now have operational pieces of the program, so we are ready to experiment more. Chapter 3, Choosing the Best Constructs, presents some tools and techniques for program development. Once the little pieces are working well independently, we will tackle the more difficult problem of making them interact smoothly. Chapter 4, Putting the Pieces Together, presents some strategies for this step of the project.

Chapter 3

CHOOSING THE BEST CONSTRUCTS

3.1 STEPS TOWARD GOOD CODE

Once you have an approach that makes sense, that is well organized, and that has been tested in sample code, you can focus on the more concrete aspects of coding. This chapter presents some ideas to help you write clean, understandable, and efficient code.

- Section 3.2, Lisp Data Structures, explains the predefined means of representing data in Common Lisp and describes situations in which certain data structures have advantages over others. This information can help you choose data structures to suit your program.
- Section 3.3, Appropriateness of Lisp Constructs, compares some commonly used Lisp constructs and suggests contexts in which one construct might be more effective than another. In this discussion, effectiveness is judged by an expression's impact on the readers and maintainers of the code, not by run-time behavior. The intent of this section is to explore the expressive power of Lisp by examining a small subset of its functions, macros, and special forms.
- Section 3.4, Programming Idioms, describes some abbreviations and idioms that are widely accepted by the Common Lisp community. It also suggests guidelines for using idioms in your own code.

The points made in each of these sections do not stand alone. For example, the effectiveness of a particular data structure depends heavily on the code that implements it. Similarly, the connotation associated with a construct can become polluted if the data structure being operated on is an inappropriate model for the problem being solved. However, though the concepts presented here are intertwined, the sections can be read as independent units; in addition, the subsections within each section generally do not rely on surrounding material. Feel free to skip around as you read this chapter.

3.2 LISP DATA STRUCTURES

Many languages require you to make decisions about data structures, low-level representations, and protocols early on and then stick to them. Because Lisp has typed data objects and powerful abstraction capabilities, you have some leeway in deciding which representation to use for your program; that is, you can choose one representation for your initial draft, write an abstract interface to it, and change to a different representation once you understand better how the program works. Still, an initial good guess saves time and energy.

Lisp offers a variety of predefined data structures to accommodate many needs. Every data structure has *accessors* and other built-in functions that help in finding and modifying the data contained in that structure. In the following sections, we suggest when one data structure might be a more appropriate choice than another for overall effectiveness.

3.2.1 Lists

The *list* is the most basic data structure in Lisp; it represents a sequence of objects that has no numeric ordering. The most common list accessors are `car` and `cdr` and their relatives. You can make new lists with the functions `cons`, `append`, `list`, `list*`, and `make-list`. The macros `push` and `pop` add elements to and remove elements from lists, and the function `copy-list` copies top-level list structures. There are functions to find elements, substitute elements, and remove elements in lists. There are also functions to modify the internal structure of a list.

The abundance of supporting operations make lists convenient data structures for storing data. If you must accommodate an unknown number of items that you intend to modify often, lists are a good choice. However, to locate an object in a list requires a linear search, so the time required to find a list element increases linearly with the size of the list. Thus, lists are not efficient data structures for large amounts of data that require random access; vectors are better in that case. Note that lists take about twice as much storage space as vectors, since each cons cell in a list takes two words, while a vector is simply a linear chunk of memory. A *cons cell* is a data structure of two parts, a *car* and a *cdr*. In most Lisp implementations, the car is a pointer to a value in memory, and the cdr is a pointer to another cons cell or to `nil`. Vectors are described in Section 3.2.2.

There are two special types of lists—association lists and property lists. Most of the general list operators work with association lists and property lists; however, list operators, such as `member`, that only work with top-level lists are not as useful as the specialized operators for association lists or property lists, such as `assoc` or `get`.

An *association list* is a list of conses such that the car of the cons specifies a key and the cdr associates a value with that key. The function `assoc` accesses elements of an

association list, and you can make or modify association lists with the functions `acons` and `pairlis`.

An important feature of association lists is that you can change the value associated with a key by simply adding the new key/value pair to the front of the list. The old value will still exist on the list, but when `assoc` searches for the key, it will stop at the first match it finds. However, `assoc` still must search linearly to find any item. Thus, association lists are reasonable data structures for storing small to moderate amounts of data that might be regularly updated. The amount of data that can be efficiently handled by association lists varies across implementations; in Lucid Common Lisp Version 3.0, the cutoff point at which the time to search for an element in an association list is longer than looking up the symbol in a hash table is eight elements.

Every Lisp symbol has a property list (or *plist*) component. A *property list* is a list of attributes of the form (*indicator value* ...), where *indicator* gives the name of the attribute. Because these attributes identify the symbol in some way, by convention each attribute has a unique indicator. The functions `get`, `getf`, and `get-properties` access elements of a property list, and the function `remprop` and the macro `remf` destructively remove elements. The function `symbol-plist` returns the property list for any given symbol.

Property lists are generally used for information about symbols that can change dynamically. You do not have to specify what the property list for a symbol will look like ahead of time, so it is a flexible recording mechanism. You can store information of any type in a property list, including function definitions. In fact, in the days before Common Lisp, property lists often held the information now stored in independent components of a symbol—its print name, value, and function definition.

As with the other types of lists, linear searching is required to locate any symbol on a property list. Thus, limit your use of property lists to small amounts of data. In addition, be aware that property lists are not first-class data objects; they suffer from the following constraints:

- The property list of a symbol is a global resource; other programs are free to use a symbol's property list at will. You cannot rely on the contents of a property list remaining static, and if others use a property list often, you cannot predict the access time for any property list element.
- Binding a symbol's value cell does not affect its property list, as can be seen in the example that follows:

```
> (setf (get 'my-symbol 'weight) 160)
160
> (symbol-plist 'my-symbol)
(WEIGHT 160)
> (setq my-symbol 'rosco)
ROSCO

;; LET binding rebinds the symbol value slot of a symbol;
;; it does not affect the property list.
> (let ((your-symbol my-symbol))
     (inspect 'your-symbol))
#<Symbol 95BF8F>

[0: NAME] "YOUR-SYMBOL"
[1: VALUE] ROSCO
[2: FUNCTION] Undefined
[3: PLIST] NIL
[4: PACKAGE] #<Package "USER" 4B5216>
>> :q
YOUR-SYMBOL
```

Though lists become less useful as the size of your database grows, they are often handy data structures for the early development stage of a program because they are so easy to manipulate. If you are careful to separate the levels of abstraction within your program by building abstraction barriers, you can easily switch to a more complex data structure once your program operates effectively using lists. Remember to write your code so that switching data structures is easy and you are not locked into a certain data structure. See Chapter 4 for information about using abstractions and abstraction barriers.

3.2.2 Arrays and Hash Tables

Arrays and hash tables help you to manage large amounts of information. *Arrays* organize data by integer indexes, and *hash tables* organize data by a compact encoding of a Lisp object.

Lisp arrays are linear data structures that are similar to array data types in other languages but with several useful enhancements. First, you do not have to specify the types of the array contents—you can mix data types in array elements. However, you can specify the types of the elements as advice that the compiler can use for optimization. Lisp arrays can be *adjustable*; that is, they can grow dynamically. Lisp arrays can be *displaced* to other arrays to share data. Single-dimensional arrays, or *vectors*, can have a fill pointer

to indicate how many of the elements of the vector are active. Access to elements of simple arrays can be optimized by Lisp compilers; a *simple array* does not have a fill pointer and is neither adjustable nor displaced. There are two special types of vectors: strings are vectors of type `string-char`, and bit-vectors are vectors of type `bit`.

The functions `make-array` and `vector` create Lisp arrays, and the function `adjust-array` modifies the length of adjustable arrays. The functions `aref`, `svref`, `char`, `schar`, `bit`, and `sbit` access elements of arrays. In addition, there are predefined data type and array predicates and, for bit arrays, predefined logical operators.

Arrays are useful for storing information that you can access by an integer index. Lisp strings and input/output buffers are good examples of the types of data that work well in arrays. Each character in the string occupies consecutive elements of the array; to get the entire string, you repeatedly access as many characters as the length of the string. When you access only one position in the string, a single character is returned.

Arrays are not effective when elements might often be inserted into the interior of the data structure rather than added to the end. Every time you insert an element into an array, you must shift the indices of all subsequent elements. In such a situation, a list is preferable to an array.

For vectors, it takes constant time to access an element, but searching for an element is a linear operation. The access time increases with the dimensions of the array. For very large amounts of data, a hash table might be a more efficient data structure than an array if the data can be cast in the appropriate manner.

Lisp hash tables organize data by a built-in encoding system that associates a unique identifier for every entry. The same algorithm that hashes an object locates that object in the hash table. Hashing is the best choice for storing large amounts of data that you need to access often and quickly, since hash tables have close to a constant access time.

The function `make-hash-table` creates a Lisp hash table. You can specify the initial size of a hash table and how it should expand when more information is added than the initial size can accommodate. Growing a hash table requires *rehashing*; that is, it requires that the key for every object be reencoded. It is best to avoid rehashing because it affects every element in the table and thus can be a time-consuming operation. The function `gethash` locates a symbol in a hash table. You can use the macro `setf` with this function to modify the contents of the table or to add new elements. The function `remhash` removes elements from a hash table.

3.2.3 Structures and Classes

Structures and classes are Lisp data structures that consist of a series of *slots*, which are simply attributes of a structure or class that can adopt values. These data structures are

suited for grouping together pieces of information about a single object. When you create a structure or a class, you define a new Lisp data type, and an object of this data type is called an *instance* of the structure or class. Lisp structures are analogous to Pascal records and C structs.

The macro `defstruct` creates a Lisp *structure* whose name designates a new data type. A fixed number of structure slots define the data type, and each slot can contain any Lisp data type, including one defined by a previous structure. You do not have to specify the type of a Lisp structure slot as you do with Pascal records and C structs. However, you can specify the type of a slot for optimization purposes. In addition, whenever you create a `defstruct` type, Lisp automatically defines access and update functions for every slot of the type, a means for making instances of that type, a predicate that identifies objects of that type, and a copier function that returns a copy of the new structure type.

You can also specify a number of options when defining a structure. These options allow you to write your own accessor, constructer, copier, and predicate functions to replace the ones that are automatically defined (see Steele for a description of the more obscure options of `defstruct`). You can also specify that one structure type includes the slots of another structure type; in this way you can define a type hierarchy.

```
> (defstruct person name age sex)
PERSON
> (defstruct (husband (:include person))
    wife)
HUSBAND
> (defstruct (father (:include husband))
    children)
FATHER
> (setq george (make-father :name 'george
                            :age  45
                            :sex  'male
                            :children '(elroy judy)
                            :wife 'jane))
#S(FATHER NAME GEORGE AGE 45 SEX MALE WIFE JANE CHILDREN (ELROY JUDY))
```

Each structure can include at most one other structure—multiple inheritance of structures is not possible.

This extensive support mechanism makes structures useful for defining complex objects with a fixed number of constituent parts that can be identified and accessed. Note, however, that you cannot dynamically add slots to a Lisp structure; the number of slots for any structure is fixed.

CLOS classes are similar to structures in many ways but are more powerful. (In fact, structures have been redefined to conform to the class system established by CLOS. In

effect, structures are second-class classes.) For example, classes support *multiple inheritance*; that is, one class can simultaneously inherit the slots and behavior of several otherwise unrelated classes. Thus, the class hierarchy can be multidimensional. For every slot contained in a class definition, there are a number of slot options. These options can associate reader and writer methods with the slot, can specify how the slot is stored, can provide initial values or documentation for the slot, or can specify the data type of the slot contents. In addition, you can define generic functions and methods that operate on instances of class data types. Generic functions can work for any objects that have methods whose parameters agree with the generic function.

The macro `defclass` defines a new class and returns a class object. The generic function `make-instance` creates an instance of a particular class. The macros `defgeneric` and `defmethod` are the common means of defining generic functions to operate on classes. The macro `defmethod` defines methods for performing operations on an instance in response to generic function dispatch. In addition, CLOS provides ways of combining methods for greater program modularity and flexibility (see Chapter 4).

In most CLOS implementations, using `defclass` is a more expensive operation than using `defstruct`, since `defclass` allows some features, like multiple inheritance, that are difficult to provide efficiently. Some implementations allow you to limit the flexibility of `defclass` to improve its efficiency.

3.3 APPROPRIATENESS OF LISP CONSTRUCTS

Often the choice of which Lisp construct to use in an expression is complicated by the existence of several similar but subtly different constructs, each of which can get the desired response from the evaluator. You are no doubt familiar with such problem constructs in English. For example, consider the relative pronouns *that* and *which*. In practice, these pronouns are sometimes used interchangeably, though *that* is meant to introduce restrictive (essential) clauses and *which* begins nonrestrictive (descriptive) clauses.

Suppose we have made three apple pies and want to tell you about them. If we say, "We used pippin apples in the pie, which is on the kitchen window sill," we have communicated to you some general information about our pie-baking technique, but a nuance about the pies is lost. You might assume that we had made only one pie, since the relative pronoun *which* implies that the location of the pie is incidental to its identification. (You cannot hear the comma.) If instead we say, "We used pippin apples in the pie that is on the kitchen window sill," the full import of the statement becomes clear. You know that the window-cooling pie is one pie we made, and you can infer from our use of *that* that there are other pies (elsewhere) to tempt you. Thus, by using the more appropriate pronoun, we convey more information to you, or we convey information to you more effectively.

The sections that follow examine Common Lisp constructs that have essentially the same meaning but different connotations to the human reader. The Lisp system is not bothered by these subtleties, but the person who wants to modify or maintain the code might be. The discussion proceeds from the most commonly used constructs to the more esoteric.

3.3.1 Conditional Constructs

A *conditional expression* is a form that consists of a conditional construct, one or more expressions serving as test conditions, and one or more expressions representing action that depends on the test. The *consequent* of a condition is the action that results when a test succeeds. The most common conditional constructs are `when`, `unless`, `if`, `case`, and `cond`, in order of increasing generality:

- The macro `when` performs a set of actions whenever its test condition succeeds. If the test fails, no actions are taken and `when` returns `nil`.
- The macro `unless` is the opposite of `when`. It performs a set of actions when its test condition fails. If the test succeeds, no actions are taken and `unless` returns `nil`.
- The special form `if` allows alternative actions to proceed from one test—one action if the test succeeds and the other one (called the *else* form) if the test fails. (It is possible to execute a series of actions as a result of the test if you create a block; see Section 3.4.2.)
- The macro `case` executes a set of actions when the result of evaluating a key matches one of the listed test forms.
- The macro `cond` accepts several test conditions and executes a set of actions when one of those tests succeeds.

Steele notes that the choice of which conditional form to use in any situation is a matter of taste and style. Though you can write `if` conditionals as `cond` conditionals, `when` and `unless` conditionals as `if` conditionals, and `case` conditionals as `cond` conditionals, the following guidelines generally apply:

- There are several options to consider when an `if` expression does not have an *else* form:
  - Don't rely on `nil` as a default return value. To return an empty list, provide an *else* clause that is `'()`.

- If you want a non-`nil` value to be returned as a result of the test, rewrite the expression to use `and` instead of `if`. However, be careful not to use `and` gratuitously as a conditional operator if a return value is not required (see the following discussion of `and`).
- When the consequent is not important as a return value but rather as a side effect, rephrase the conditional to use `when` or `unless` instead of `if`.

- If you have one test and two alternative actions that can result from it, use `if` rather than `cond` or `case`:

```
;;; This form works, but it's more than is necessary.
(defun test (color)
  (cond ((eq color 'pink) (format t "The rose is ~S" color))
        (t (format t "I cannot see the color of the rose."))))

;;; This form works, but it's awkwardly unnecessary.
(defun test (color)
  (case color (pink (format t "The rose is ~S" color))
              (otherwise
                (format t "I cannot see the color of the rose."))))

;;; This form's just right.
(defun test (color)
  (if (eq color 'pink)
      (format t "The rose is ~S" color)
      (format t "I cannot see the color of the rose.")))
```

- If you have many conditions to express, use `cond`, which provides the same functionality as consecutive *if-else* statements in such languages as C or Pascal.
- If you want to compare a single value against a number of constants and take a different action depending on which constant matches, use `case` instead of `cond`.
- If the conditional test is resolved by comparing strings or lists, `case` will not work, since it uses `eql` to determine when a key is matched. In such instances, use `cond` instead of `case`.

Beware of deeply nested conditional expressions. If you need a road map to trace the behavior of a nested conditional, just think of the problems someone not familiar with your reasoning might have following it. Consider breaking up the problem into smaller pieces, each of which provides one level of the test.

The discussion of style for conditionals is not complete without considering the special last test for `cond` constructs, a `t` test. There are three good reasons for including in a `cond` a final test that is guaranteed to succeed:

- For robustness it is good practice to take some action if all else fails, including, with recursive functions, calling the function again with different data.
- For clarity of intent, if the function needs to return `nil` when no other action is taken, explicitly include `(t nil)` as the final clause instead of relying on the default behavior of `cond`.
- A `t` test in the final clause ensures that multiple values can be returned from that clause. If the last clause of a `cond` is not part of a `t` test, at most one value can be returned if that clause is reached.

3.3.2 Logical Operators (and a Few More)

The macros `and` and `or` and the function `not` correspond to the three basic logical operations of Boolean algebra—*and*, *or*, and *complement*:

- The macro `and` evaluates its arguments from left to right until one of the arguments evaluates to `nil`, in which case it returns `nil`, or until it evaluates its last argument, in which case it returns the result of that evaluation.
- The macro `or` evaluates its arguments from left to right until it finds one that has a non-`nil` value, in which case `or` returns that value, or until it has evaluated all the arguments, in which case it returns `nil`.
- The function `not` returns `t` if the value of its argument is `nil`; otherwise, it returns `nil`.

You can use `and` and `or` in non-Boolean contexts with good results if you exercise stylistic restraint. As noted, `and` and `or` do not evaluate arguments past the one that satisfies their contracts. You can use this behavior to control the evaluation of certain expressions. For example, you can include a type-checking expression as the first argument to `and` to check the type of an operand that is used in a subsequent operation or operations:

```
(and (integerp x) (integer-length x))
```

If the argument `x` is not an integer, `and` returns `nil`; no type error is signaled because `integer-length` is never invoked. If `x` is an integer, `and` returns the result of the last expression.

The evaluation of every argument to `and` should provide meaningful information for continuing the evaluation; that is, interior expressions in `and` clauses should not produce

side effects. If you are interested in producing side effects, switch to a `when` construction, since `when` has an implicit `progn` and is intended to produce side effects rather than to return values.

You can use `or` in a similar type-checking context to exclude a certain data type from being used by an operation. However, if only side effects are desired, use `unless` instead.

The functions `not` and `null` actually perform equivalent operations, but they imply different intents. Use `null` only in reference to lists—that is, to test for the empty list, `()`. In contexts of logical inversion, `not` has the proper connotation.

You can also use the function `endp` to test for the end of a list, especially during list iteration. Whenever `endp` receives a non-`nil` argument that is not a cons, it signals an error.

3.3.3 Cons and List Accessors

Every Lisp programmer is familiar with the functions `car` and `cdr`, which access the components of a cons. These functions can be combined to form 28 different Common Lisp functions, each of which accesses a different component of the cons. This composition of `car` and `cdr` is convenient, but sometimes it is not the clearest means of expressing the operation that it performs. For example, how long does it take you to decipher `cddadr`? "Let's see, that's the cdr of the cdr of the car of the cdr of the cons. So I take the cdr and get its car, which has to be a cons. And then I take the cdr of that cons, and then I take the cdr again."

Suppose you need to get information from a property list. If you know exactly how the information is stored, you can figure out which composition of `car` and `cdr` will reach the information you need. However, you can make your program more readable by giving that composition a more descriptive name within a function definition; that is, you can write an abstract accessor function. This approach has the added advantage that only one function will need to be rewritten if the data representation is ever changed.

```
;;; The employee information is stored in an association list.
> (setq employee
        '(name . ((street-number street-name city state) phone)))
(NAME (STREET-NUMBER STREET-NAME CITY STATE) PHONE)

;;; Here's one way to get the city of residence, if you can
;;; remember how the information is stored.
> (cddadr employee)
(CITY STATE)
```

```
;;; If the data representation changes, you must change the
;;; definition of this function, but the user retrieves the
;;; information in the same mnemonic way.
> (defun hails-from (person)
    (cddadr person))
HAILS-FROM
> (hails-from employee)
(CITY STATE)
```

The same advice applies to the group of accessors associated with the functions `first` and `rest`, which are the accessors used with lists. For example, hide a call to `sixth` inside a function with a name that more clearly describes the operation being performed on the particular list element being accessed. It might be better to reconsider the choice of data structure if you must contrive abstract accessors to match awkward list representations; consider whether a structure, with its built-in accessors, might be more appropriate. See Section 3.2.3 for a description of structures; Section 4.2 gives a more complete discussion of abstraction techniques.

Note that `first` is equivalent to `car`, and `rest` is equivalent to `cdr`. To make the programmatic distinction between cons access and list access, use `car` and `cdr` with pairs, and `first` and `rest` otherwise, as the following example shows:

```
;;; This function performs the same operation as
;;;                 (cdr (assoc item in-list :test #'eq)).
;;; FIRST and REST are used for list access, while CAR and CDR are
;;; used for cons access.  An association list is a list of conses.
(defun find-eq-entry-in-alist (item in-list)
  (if (null in-list)
      nil
      (let ((entry (first in-list)))
        (if (eq (car entry) item)
            (cdr entry)
            (find-eq-entry-in-alist item (rest in-list))))))
```

3.3.4 Assignment Operators

The most general of all the Common Lisp sequential assignment operators is the macro `setf`. It works on lexical as well as dynamic variables, certain function call forms, and `the` type declarations. The special form `setq` is simpler than `setf`; it works only with lexical and dynamic variables.

You can use `setf` in every case that allows `setq`. However, if you know that the particular assignment always applies to variables, you can advertise your intent by using `setq`.

Whenever possible, use the macro `defvar` or `defparameter` to create special or global variables. See Section 3.3.7 for more information about these macros.

The macro `psetf` works like `setf` except that it assigns values in parallel, while `setf` assigns values to its arguments sequentially. The special form `psetq` is the parallel version of `setq`.

You can use the macro `setf` programmatically to good purpose for updating data structures through their accessors. The next section describes this process in detail.

3.3.5 Access and Update Operations

Many types of data storage are available in Common Lisp, as described in Section 3.2. For each type of data structure, you could write specialized constructs that both access a location of a data type and update its value, like the Common Lisp functions `rplaca` and `rplacd`, which respectively update the car and cdr of a cons. For example, you could write a function named `putprop` that changes the value of a specified indicator in a property list:

```
(defun putprop (atom property value)
  (setf (get atom property) value))
```

In addition to the amount of program space these specialized constructs would take, you might have a hard time remembering the names of all of them. Luckily, the macro `setf` provides a consistent approach to updating named storage locations. (As you can see from the example, `setf` would be the underlying construct to the specialized constructs anyway.) You can use `setf` with any of the predefined Common Lisp accessors, as well as with homemade accessor functions, to assign a new value to the data stored in the accessed location. To preserve the consistency of this approach, use `setf` even with `car` and `cdr` in place of the cacophonous `rplaca` and `rplacd`.

3.3.6 Variable Bindings

The special forms `let` and `let*` establish variable bindings. Generally, `let` bindings are lexical. For this reason, `let` and `let*` are the preferred means of introducing temporary, or local, variables into Lisp expressions. Variables created this way are restricted in use to the construct that establishes them.

Note that `let` bindings occur in parallel order. The order in which updates occur is not important when the value of one variable in a `let` form does not depend on the updated value of another of the `let` variables. But for cases involving dependent values, use `let*`, which binds its variables to values in sequence. The following example from the *Personal Planner* illustrates these differences:

```
(defun menu-member-acceptor (type plural-string)
  (format t "~&Select one of the following ")
  (when plural-string
    (write-char #\Space)
    (write-string plural-string))
  (format t ":~%")
  (let* ((choice-list (rest type))
         (number-of-choices (length choice-list)))
    (loop for i from 1
          for choice in choice-list
          do (format t "~&~D: ~S~%" i choice))
    (let ((choice (accept `(or (integer 1 ,number-of-choices)
                               (member ,@choice-list))
                          "number or name" "numbers or names")))
      (if (integerp choice)
          (nth (1- choice) choice-list)
          choice))))
```

In the first lexically bound expression, we need `let*` so that we can use `choice-list` after it is bound to `(rest type)`; in the other lexical expression, we have no binding restrictions, so we use `let`.

If you want the initial value of a `let` variable to be `nil` or the empty list, include an explicit initialization expression at the beginning of the `let` form rather than relying on a default value. This precaution makes your intentions clear. For similar reasons, if you do not intend to use the initial value of the `let` variable inside the lexical expression, do not supply an explicit initial value.

Use variable binding over assignment wherever it is feasible. Since most variable bindings are lexical, the scope of the binding is limited to the location of the construct that contains it. No construct outside of the scope can reference that binding or be affected by it; the value of the variable is not associated with a symbol. Thus, by using lexical bindings, you can more closely control the execution of the program. A useful benefit of this limited scope is that someone reading the program construct can tell what values the variables are meant to have by looking at a small portion of code. This feature, called *local transparency*, makes the program easier to understand and thus easier to debug and maintain (Brooks, p. 95).

3.3.7 Constants, Parameters, and Global Variables

How often have you seen expressions like this buried deep in the middle of some complex function definition:

```
(let ((interval (* 24 60 60))) ...)
```

What does this definition of `interval` mean? Why these numbers and not others? What happens if the interval needs to grow or shrink; that is, what are the units of the interval? If you think about this definition briefly, it makes sense; however, it is still not explicit. This situation begs for a symbol, and in this case the symbol can also be a constant, which is defined by the macro `defconstant`:

```
(defconstant seconds-per-minute 60)
(defconstant minutes-per-hour 60)
(defconstant hours-per-day 24)
(defconstant seconds-per-hour (* seconds-per-minute minutes-per-hour))
(defconstant seconds-per-day (* hours-per-day seconds-per-hour))
(defun passing-of-time (day week year)
  (let ((interval seconds-per-day))
    ...))
```

The constant value makes the last expression easier to read and understand. Whenever you use one of these symbols, such as `seconds-per-minute`, in another expression, it is clear what is being represented; it would not be so clear if you simply included the value 60. Also, the compiler knows from the outset about the name. For example, it can assume that the value of `hours-per-day` will not change.

The intent of a constant is to represent global values that are not expected to change. You might have some global values that are fairly stable but whose definitions depend upon the environment in which they are run. You can introduce such dependent values, or *parameters*, with the macro `defparameter`.

For example, suppose you wanted to define some global symbol types. You have a fairly clear picture of the ones you need for now, but you might find that you need to define others as the program becomes more widely used. You could define these types in an association list that is a global parameter, as follows:

```
(defparameter *program-types*
        '((ratio . #(numerator denominator))
          (complex . #(realpart imagpart))
          (single-float . #((mantissa . decode-float)
                            (exponent . inspect-exponent)
                            (sign . float-sign)))))
```

This parameter is accessible by every part of your program. And when you revise the program to support a new type, you simply update the new global parameter definition:

```
> (setq *print-array* t)
T
> (defparameter *program-types*
    (cons '(mytype . #()) *program-types*))
*PROGRAM-TYPES*
> (pprint *program-types*)
((MYTYPE . #())
 (RATIO . #(NUMERATOR DENOMINATOR))
 (COMPLEX . #(REALPART IMAGPART))
 (SINGLE-FLOAT . #((MANTISSA . DECODE-FLOAT)
                   (EXPONENT . INSPECT-EXPONENT)
                   (SIGN . FLOAT-SIGN))))
```

Note that the compiler cannot make assumptions about global parameters defined with `defparameter` as it does with `defconstant`; even though a global parameter is intended to be fairly static, the compiler must assume that it might change at any time.

The macro `defvar` defines global variables that most likely will change over the course of the program. Consider the global variable `*command-table*` in which the *Personal Planner* stores user-input commands. Every time a new command is defined, the function `add-command` updates the list contained in this variable:

```
(defvar *command-table* '())

(defun add-command (command-names user-documentation function)
  ;; Remove any existing commands of the same name.
  (dolist (table-entry *command-table*)
    (dolist (existing-command
             (command-table-entry-names table-entry))
      (when (member existing-command command-names :test #'eq)
        (setf (command-table-entry-names table-entry)
              (delete existing-command
                      (command-table-entry-names table-entry)
                      :test #'eq))))
    ;; If we removed all the names for this command, remove the command.
    (when (null (command-table-entry-names table-entry))
      (setq *command-table*
            (delete table-entry *command-table* :test #'eq))))
  ;; Add the new command to the table.
  (setf *command-table*
        (nconc *command-table*
               (list (make-command-table-entry
                      :names command-names
                      :documentation user-documentation
                      :function function)))
```

```
;; Remove existing names from the names list.
(dolist (new-name command-names)
  (setf *command-list*
        (delete new-name *command-list* :test #'eq)))
;; Add the command names to the names list.
(setf *command-list*
      (nconc *command-list* (copy-list command-names)))
command-names)
```

In this example, `*command-table*` is a list of structures, each of which consists of a name, a documentation string, and a function object that performs the action specified by the command. The function `add-command` first loops through the list `*command-table*` to see if any of the existing commands have the same name as the one being added; if so, it removes the existing entry. It then pushes the new command structure onto the global list of commands.

Note that you cannot use `defvar` to change the value of a global variable that already has a value; this restriction does not apply to `defparameter`:

```
> (defparameter *temp* 10)
*TEMP*
> (defvar *temp* 20)
*TEMP*
> *temp*
10
> (defparameter *temp* 30)
*TEMP*
> *temp*
30
```

Thus, if you have assigned a value to a global variable with `defvar` and you load in a file that contains a new `defvar` definition for the same variable, you will not get the expected results; the old `defvar` value will persist.

The compiler recognizes globally defined variables and parameters as being proclaimed `special`; that is, they are dynamically scoped. When you bind a special variable inside a lexical expression, the dynamic value is maintained and can be referenced and changed by constructs within the dynamic scope of the variable.

You can also create special variables by proclaiming or declaring them as special and then assigning them values with `setf` and `setq`. However, there are a couple of reasons why `defvar` and `defparameter` are preferred for creating special variables. First, `defvar` and `defparameter` both declare a variable as special and set its value. Second, variables defined with `defvar` and `defparameter` can have attached documentation of

type `variable` that can be retrieved by the function `documentation`. The documentation serves as yet another advertisement of the global nature and purpose of the variable.

We suggest you use special variables only when the value they hold is needed by many functions at many levels, such as for a database; never define a global variable that is used by a single function. For internal communication between procedures, pass arguments as parameters instead of using global variables. Whenever you define a global variable, delimit its name with asterisks to call attention to its special nature.

In general, global variables violate the principle of local transparency and thus make it more difficult for someone else to follow your program. Specials also incur run-time overhead; the code must remember the dynamic value throughout any special binding of the variable.

3.3.8 Simple Iteration

Many years ago, Lisp 1.5 introduced `prog`, which could be used with `go` to emulate certain constructs in Fortran and Algol. Over time, programmers have adopted more elegant control structures, but `prog` and `go` have remained part of the language for historical reasons. For stylistic reasons, other conditional and iterative control structures are much easier to write and to follow. Section 3.3.1 presented commonly used conditional constructs. This section discusses simple iterative constructs.

The original version of the Common Lisp `loop` macro was quite simple; it executed its body repeatedly until it was explicitly halted, as by a call to `return`. In 1989, the Common Lisp committee approved an expanded form of `loop` that lets you specify the parameters of the iteration. With the new `loop`, you can specify the direction of iteration, accumulate values inside the loop, and evaluate expressions that precede or follow the body of the loop. Though some people object to the syntax used in the revised `loop` macro, it provides a wide range of control, as the following example from the *Personal Planner* shows:

```
(defun find-type-specific (type database)
  (loop for entry in database
        as entry-type = (car entry)
        with best-entry = nil
        and best-type = 't      ;The top of the type hierarchy
        do
```

```
;; An exact match is always taken.
(when (or (equal type entry-type)
          ;; If the CAR of type matches entry-type,
          ;; e.g., for MEMBER or INTEGER, it's also okay.
          (and (consp type)
               (eq (car type) entry-type)))
  (return (cdr entry)))
;; Otherwise, check for subtypep.
(when (and (type-specifier-p entry-type)
           (subtypep type entry-type))
  ;; If it is, check if it is more specific than best-type.
  (when (subtypep entry-type best-type)
    (setq best-type entry-type)
    (setq best-entry (cdr entry))))
finally (return best-entry)))
```

This `loop` expression contains several *loop keywords* that allow us to initialize and update loop control variables in the prologue, execute the iteration body, and specify a final action in the loop epilogue. The range of control is much broader than is available with `do` and its relatives; thus, `loop` is more appropriate for complicated iteration expressions.

The macros `do`, `do*`, `dolist`, and `dotimes` are commonly used iteration mechanisms:

- `do` and `do*` perform general iteration; the index variables of a `do` construct are bound in parallel, while `do*` binds its index variables sequentially.
- `dolist` iterates over the elements of a list.
- `dotimes` iterates over non-negative integers in order, starting with zero and ending at a specified upper bound.

Sometimes it is not necessary to include the body of a `do` expression to satisfy the iterative algorithm; the updates and end test are sufficient:

```
> (defun list-avg (numlist)
    (do ((temp numlist (rest temp))
         (avg 0 (+ avg (first temp))))
        ((null temp) (/ avg (length numlist)))))
LIST-AVG
> (list-avg '(1 2 3 4 5 6 7 8 9))
5
```

Omitting the body in `dolist` and `dotimes` expressions renders them meaningless.

When you need to iterate over integers and to collect integer values, use `dotimes` instead of `do`. Besides giving type information to the compiler, this construct advises those

reading and maintaining the code that you are performing numeric iteration—it is a form of self-documentation. Similarly, when iterating over the elements of a list, you convey more information by using `dolist` instead of `do`. In the following example from the *Personal Planner*, `command-table-entry-names` is a slot accessor for the list of names associated with any command entry in the command table:

```
(defun find-command (user-entry)
  (dolist (table-entry *command-table* nil)
    (when (member user-entry (command-table-entry-names table-entry)
                  :test #'eq)
      (return (command-table-entry-function table-entry)))))
```

Readers of this code can infer from the use of `dolist` that `*command-table*` is a list.

3.3.9 Mapping Operations

Common Lisp allows another form of iteration through *mapping*, in which a function is applied to successive elements of a sequence. The mapping functions that operate specifically on successive elements of a list are `mapcar`, `mapc`, and `mapcan`:

- `mapcar` accumulates the results of the mapping and returns them as a list.
- `mapc` maps an operation without accumulating the results of the mapping. If you are interested in the side effects of the operation instead of the results, use this construct instead of `mapcar`; it is more efficient.
- `mapcan` uses `nconc` to return a concatenated list of the results of the mapping. If the mapping function returns `nil`, nothing is added to the list.

Other mapping functions have parallels with recursion in that they operate on successive cdrs of lists:

- `maplist` accumulates the results of the mapping and returns them as a list.
- `mapl` maps an operation for its side effects without accumulating the results of the mapping. This construct can be more efficient than `maplist`, so use it when the return values are not important.
- `mapcon` uses `nconc` to return a concatenated list of the results of the mapping.

The functions `maplist`, `mapl`, and `mapcon` are rarely used, since successive cdr mapping is not often required in practice.

List manipulation is such an integral part of Lisp programming that mapping is often the most natural means of expressing a certain operation. For example, it is very often

the case that you need to apply a certain function to one or more lists, and mapping does just that. The syntax of mapping is also sufficiently clear to make a mapping operation usually more straightforward to understand than a recursive expression or a `do` iterative construct.

In addition, the mapping functions are often used to execute functional lambda expressions (see Section 3.3.11 for information about `lambda`). The combination of a mapping function and a lambda expression has the added advantage of local transparency; the reader knows exactly which operation is being mapped over the list—the operation whose definition is given in full as the argument to the mapping function:

```
(defun increase-elements (increment numlist)
  (mapcar #'(lambda (x) (+ increment x)) numlist))
```

The function `map` is a more general mapping function that operates on sequences. The behavior of this function can emulate `mapcar`, when it is passed a non-`nil` *result-type* argument, or `mapc`, when *result-type* is `nil`.

3.3.10 Local Functions and Macros

If a function is only called from within another function, it is advisable to recast the operation in terms of a local function or macro. The special forms `flet` and `labels` define locally named functions. The local functions defined by `labels` can be mutually recursive or self-recursive. The following example shows a self-recursive `labels` construct, one with a single locally named function that calls itself:

```
(defun add-event (event event-time)
  (let ((tree *event-table*))
      (labels
          ((add-to-tree (event event-time tree)
             (let ((leaf-time (binary-tree-key tree)))
               (cond ((< event-time leaf-time)
                      (let ((left (binary-tree-left-branch tree)))
                    (if (null left)
                        (setf (binary-tree-left-branch tree)
                              (make-binary-tree :key event-time
                                                :entries
                                                  (list event)))
                        (add-to-tree event event-time left))))
```

```
((< leaf-time event-time)
 (let ((right (binary-tree-right-branch tree)))
   (if (null right)
       (setf (binary-tree-right-branch tree)
             (make-binary-tree :key event-time
                               :entries
                                 (list event)))
       (add-to-tree event event-time right))))
(t              ;leaf time equals event time
 (push event (binary-tree-entries tree))))))
(add-to-tree event event-time tree))))
```

Names of temporary functions established by `flet` are not visible within the body of the function defined by the `flet` expression, though names of `labels` functions are apparent within the body of the function defined by `labels`. Thus, you can reuse a global function name within an `flet` expression (except for Common Lisp function names); however, it is best to rely on this feature only in situations that would not otherwise confuse readers of your code.

The special form `macrolet` defines locally named macros.

To avoid having to remember the complicated syntax of `do`, you can use `labels` with *tail recursion* instead of iteration. In tail recursion, the value of the recursive call is the value returned by the function.

Just as with local variables created with `let` and `let*`, no symbols are associated with the temporary functions except in the lexical environment in which they are constructed. By definition they also adhere to the principle of local transparency.

3.3.11 Anonymous Functions

The macro `defun` and the special forms `labels` and `flet` all define named functions. A *lambda expression* defines a function that does not have a name. You can use a lambda expression anywhere that you would otherwise use a function, such as an argument to `apply` or `funcall`. (See Section 3.3.13 for information about `apply` and `funcall`.) In fact, lambda expressions are commonly used as arguments to other functions.

Consider the Common Lisp function `maphash`. It calls its function argument on the key and the value of each entry in the specified hash table. You can insert a lambda expression of two arguments, a key and a value, to perform the desired operation on the hash table from within the contract established by `maphash`. For this reason, lambda expressions are commonly used to express simple mapping operations, as in this example, which maps over the elements of a hash table and prints the keys and values:

```
(defun printhash (hash-table)
  (maphash #'(lambda (key value)
               (format t "~&~A: ~A~%" key value))
           hash-table))
```

A lambda expression creates an anonymous function, so you cannot refer to a lambda expression by name; instead, you pass lambda expressions as arguments to other functions. Using a lambda expression thus allows a great deal of programmatic control as well as flexibility. In addition, all lambda expressions conform to the principle of local transparency; the entire operation is available in line instead of through a call to a function at a different location.

3.3.12 Three Examples for Comparison

The following examples generate a list of random numbers and then normalize that list. Two of these examples are iterative, while the third uses a recursive `labels` function. The first example uses `dotimes` iteration to create the list of numbers and `let` binding and mapping to normalize the list:

```
;;; Generate a random list of numbers, and normalize the list
;;; so that the sum of all the numbers equals zero.  The parameter
;;; N specifies the number of elements to list, and the parameter
;;; MAX-VALUE is the upper limit for the value of any element on
;;; the list that is generated.  The lower limit is in the range
;;; (- (/ (- max-value) 2) (/ max-value 2)).
(defun generate-normalized-random-list (n max-value)
  (let ((sum 0)
        (list '()))
    ;; Build up a list of random numbers and keep track of the sum.
    (dotimes (i n)
      (let ((value (random max-value)))
        (push value list)
        (incf sum value)))
    ;; Subtract the average value from each element of the list.
    (let ((average (/ sum n)))
      (mapcar #'(lambda (element) (- element average)) list))))
```

This example is easy to follow. The temporary variable `sum` is the working sum of the elements; its initial value is zero. The temporary variable `list` is explicitly initialized to `nil`. Note that inside of `let` we use the symbol name `list` for a variable without conflicting with or altering the global function definition of the Common Lisp function `list`. In Common Lisp, unlike in Scheme, functional evaluation differs from argument evaluation; thus, you can have a function and a variable with the same name without

a namespace conflict. Do not rely on this feature of Common Lisp when the resulting expression might be misread by maintainers of your program. In this instance, however, when the `let` expression is completed, the temporary variable named `list` goes away. Since `list` is the most natural name for the construct we are building with the function, using it should not confuse the readers of the code.

By using `dotimes`, we indicate that the iteration is numeric. Inside the iterative loop, *n* random values are pushed onto the temporary list, and the sum is incremented by each value in turn. Once the list is complete, we map a temporary function over the elements of the list to normalize them. We have used `mapcar` instead of `mapc` because we are interested in the return value of the operation, not in side effects. The mapping operation is specified by a temporary function defined as a lambda expression. We could have defined a new function named `normalize` as a more general facility for normalizing the elements of a list. Since we do not need that capability, we can more closely adhere to the principle of local transparency by mapping over a lambda expression, and we can save on overhead as well.

In the second example, we translate the `dotimes` operation into a `loop` iteration:

```
;;; A revision of GENERATE-NORMALIZED-RANDOM-LIST using LOOP.
(defun generate-normalized-random-list (n max-value)
  (loop repeat n
        for value = (random max-value)
        sum value into total
        collect value into denormalized-list
        finally
        (return (loop for element in denormalized-list
                      with average = (/ total n)
                      collect (- element average)))))
```

The `loop` keyword `collect` builds the list of random numbers in the opposite order from the list built in the `dotimes` iteration, but otherwise the action performed by this `loop` example mimics the first iterative definition we discussed. In the `finally` clause, `denormalized-list` is used only as an iteration control element, and the result is collected into an anonymous list structure and returned.

The third example performs the same task tail-recursively:

```
;;; A revision of GENERATE-NORMALIZED-RANDOM-LIST using
;;; tail recursion.
(defun generate-normalized-random-list (n max-value)
  (labels ((generate (count list sum)
             (if (zerop count)
```

```
                ;; All done--normalize the list.
                (let ((average (/ sum n)))
                  (mapcar #'(lambda (element)
                              (- element average)) list))
                ;; Add another element to the list.
                (let ((value (random max-value)))
                  (generate (1- count) (cons value list)
                            (+ sum value))))))
    (generate n '() 0)))
```

The `labels` construct creates a temporary recursive function named `generate` to build the list. There are two cases to consider in the recursive process, as reflected in the use of the `if` conditional. In the first case, we have finished building the list; we use the same mapping technique to normalize the list as we used in the iterative example. Otherwise, the list is incomplete; we decrement the count, add a new element to the working list, and increase the sum by the value of that element—all in the recursive call to `generate`.

If we had used an auxiliary function to perform the iteration instead of using the local function `generate`, we would have had to pass two more parameters than `generate` required. Instead the primary function definition can encompass the auxiliary function through the `labels` construct.

Which of these examples is stylistically preferable? Are the iterative solutions better than the recursive solution, or vice versa? In fact, all of these are well-written functions; none is superior to another. So, when should you use recursion, and when should you use iteration?

Recursion is well suited for problems that can be broken down into elemental tasks that can be individually solved until the overall solution is found. Recursion is often a good choice for constructing, tearing apart, and operating on elements of lists. Iteration works for problems whose solutions can be reached through a fixed number of incremental steps. Iteration commonly is used for operations on data structures of a finite size.

There can be less overhead in a well-written iterative solution than in evaluating embedded function calls in a recursive function definition. The key phrase in the previous statement is *well-written*; poorly written iterative constructs can be very expensive, since the same mistakes are repeated every time through the loop. Any iterative operation can be rewritten as a recursive operation, and tail-recursive functions can be rewritten as iterative operations. The amount of tedium that results from transforming one type of operation to another will give you a good indication of which is the more appropriate approach for the problem at hand. As a rule of thumb, recursion works better for hierarchical data structures than does iteration. And if the method for solving a problem is difficult to find, you might try a recursive solution; by breaking the problem up into its constituent parts, the

algorithm might become more clear to you and to readers of your code.

There are many other ways that we could express this function in Common Lisp. For more examples, see Section 7.2.1.

3.3.13 Function Application

The functions `apply` and `funcall` apply a function to arguments. When do you need to apply a function instead of calling a function by name? An obvious choice is to execute an unnamed function through a lambda expression. More generally, function application lets you use functions as data. Consider the following code fragment from the *Personal Planner*:

```
(defstruct command-table-entry
  (names '() :type list)
  (documentation "" :type string)
  function)

(defun evaluate-command (user-entry)
  (funcall (command-table-entry-function (find-command user-entry))))
```

Every user-visible command in the *Personal Planner* is a structure that is kept on a global list of commands. The slots of the structure contain identifying information about the command as well as a function to perform the operation that it specifies. Whenever the user types in a command, the function `evaluate-command` calls `find-command` to locate the command structure in the table; then, using `funcall`, it applies the function stored in the structure to execute the command. Failing such a direct means for carrying out user commands, the solution would be more complicated; we would probably use a `case` conditional whose keys were command names and whose consequents were calls to functions that would execute the corresponding commands. This approach gets messy when there are many commands for `case` to consider and when we need to add new commands to the system.

The purpose of `funcall` is to call a functional argument; it allows you to invoke computed functions. While `apply` has the same property of computing a function, it is distinguished from `funcall` in that it can *spread* its arguments. That is, if the last argument to `apply` is a list, `apply` takes the list apart into a number of constituent elements.

Certain functions accept an unspecified number of arguments, such as those indicated by the `&optional`, `&rest`, and `&allow-other-keys` keyword parameters. Thus, you can specify certain arguments in some cases and leave them out if they are not necessary. If you do not know in advance how many arguments you need to pass to a function in a given situation, you could write separate conditionalized invocations, each with a different

set of arguments. However, the more appropriate choice is to use `apply`, which can apply the function definition to an arbitrary number of arguments.

When you know the number of arguments that a function should take but you cannot invoke the function directly, you can apply the function with `funcall`.

As a convenience, whenever you supply a symbol as an argument to `funcall` or `apply`, the global function definition of the argument is used; however, it is better style to make your intentions explicit by not relying on this behavior and by specifying functional objects instead of symbols as arguments.

You can use these constructs to differentiate between the value of the symbol of a given name and its function definition:

```
;; Attach a function definition to a symbol.
> (defun confusing-lisp-symbols (x)
    (+ 100 x))
CONFUSING-LISP-SYMBOLS

;; Set the symbol value as well.
> (setq confusing-lisp-symbols "yikes")
"yikes"

;; You can access both slots of the symbol.
> (symbol-function 'confusing-lisp-symbols)
#<Interpreted-Function (NAMED-LAMBDA CONFUSING-LISP-SYMBOLS (X)
(BLOCK CONFUSING-LISP-SYMBOLS (+ 100 X))) 930EEF>
> (symbol-value 'confusing-lisp-symbols)
"yikes"

;; Change the symbol value to be another function definition.
> (setq confusing-lisp-symbols #'(lambda (x) (cons x x)))
#<Interpreted-Function (LAMBDA (X) (CONS X X)) 6980BF>

;; Apply the function that is in the function cell of the symbol.
> (funcall #'confusing-lisp-symbols 25)
125

;; Apply the function that is the symbol value.
> (funcall confusing-lisp-symbols 25)
(25 . 25)
```

As a final pointer, remember `apply` and `funcall` whenever you start to use `eval`. Every time you use `eval`, there is both an implicit and an explicit evaluation: normal Lisp evaluation of the expression takes place implicitly, and then the function `eval` explicitly evaluates its argument. Thus, calls to `eval` can be expensive. Also, since `eval` is just a

function, it does not know anything about the lexical context in which it is called; that is, it cannot recognize local variables. If your intention is simply to invoke functional objects, `funcall` and `apply` are cheaper and more appropriate than `eval`. If you are interested only in the dynamic value of a symbol, the function `symbol-value` is the right choice instead of `eval`.

3.3.14 Equality Predicates

Common Lisp has several equality predicates to choose from, and each provides a different level of generality. In considering these predicates, keep in mind that Lisp values are object references that have a corresponding object in memory.

- `eq` compares object identity; that is, it checks for identical Lisp object references. Two values are `eq` if they refer to the same object.
- `eql` compares values for numbers and characters and identity for all other objects.
- `equal` compares structures of lists and strings and uses `eql` for all other objects.
- `equalp` compares structures of arrays and lists, treats numbers like =, and uses `eql` for all other objects. It ignores case distinctions in strings.
- = compares numeric values and ignores type differences. It can only be used to compare numbers. During the comparison, it performs arithmetic coercion of its arguments, which can incur a certain amount of overhead.
- `char=` compares characters and respects differences in case.
- `char-equal` compares characters in a case-insensitive manner.
- `string=` compares strings character by character and respects differences in case.
- `string-equal` compares strings character by character in a case-insensitive manner.

The functions `eq`, `eql`, `equal`, and `equalp` are generic comparison operators—they work on objects of any type. The differences between `eq` and `eql` occur at the implementation level. The function `eq` offers the most stringent test and is generally the fastest of all the equality predicates, since it compares pointers. However, it is not guaranteed by the Common Lisp standard to be true for characters and numbers that have the same values.

You cannot use any of these predicates to compare equality of Lisp structures created with `defstruct`. You must write your own predicates to compare structures slot by slot.

The functions =, `char=`, `char-equal`, `string=`, and `string-equal` perform type-specific comparisons. Thus, if a comparison only involves numbers, = is more expressive than one of the generic comparison operators; readers associate expressions using =

with numerical calculation. When comparing characters, the functions `char=` and `char-equal` are not only more effective but also alert a reader of the program to the type of data you expect. Similarly, for string comparison, `string=` and `string-equal` call attention to the type of comparison taking place.

Avoid using equality predicates as a substitute for a more explicit predicate. For example, the expression `(equal lis '())` is better expressed as `(null lis)`.

3.3.15 File Accessors

As in any programming language, files are repositories for program code as well as for program data. Common Lisp provides two constructs for gaining access to files: the function `open` and the macro `with-open-file`, which is built on `open`. In fact, `with-open-file` is the most common way of using `open`.

Whenever you call `with-open-file`, a stream is opened. You can specify the characteristics of the stream and error-handling mechanisms for it by using the same keyword options that apply to `open`. However, the stream only remains open while the body of the macro executes. When the execution is complete or control is transferred elsewhere (even as a result of an error), the stream is automatically closed. Thus, `with-open-file` has dynamic extent; you must perform all input and output operations on the stream before returning from a `with-open-file` form.

```
(defvar *out-stream*
  "An output stream to which page totals are written")

(defun directions (segment-to-total filename)
  (format t "If the default value is correct, enter a carriage-return. ~
             Otherwise, enter a new value.  ~
             Any nonzero value that you supply will become the ~
             default value.~2%")
  (with-open-file (*out-stream* filename
                   :direction :output :if-exists :supersede)
    (page-thumber (cdr segment-to-total) 0)))
```

In this example, all stream operations occur in the function `page-thumber`, which is embedded in the call to `with-open-file`.

You must explicitly close any stream established by the function `open` because it has indefinite extent. Such a stream can be accessed globally by any I/O operation and remains open until the function `close` turns it off; an error cannot cause this stream to close. For this reason, `with-open-file` is safer and more convenient than `open` and is preferred when the dynamic extent of the stream is sufficient.

3.4 PROGRAMMING IDIOMS

Abelson and Sussman define programming idioms as "standard program structures of whose correctness we have become sure" (p. xii). Working with the language time and again, Lisp users have adopted certain idiomatic expressions. Some of these idioms are parts of Common Lisp, while others are personal shortcuts.

3.4.1 Common Abbreviations

The most commonly used Common Lisp idioms are abbreviations that have special reader syntax:

- The quote character (') substitutes for the function `quote` to prevent evaluation of a form. The object following the quote character is treated as a constant.

```
(in-package 'plan :use '(lisp))
```

- The combination of the dispatching macro character # with a quote character (#') substitutes for the special form `function`.

```
(defvar *database* (make-hash-table :test #'equal))
```

- The backquote character (`) combined with comma (,) and at-sign (@) characters provides a template for list construction and is often used in macro definitions. Backquote inhibits evaluation, and a comma within a backquote promotes evaluation. An at-sign combined with a comma inside a backquote indicates that the succeeding variable is a list; the result of evaluating that list will be spliced into the template at the location marked with the ,@ sequence.

The following example from the *Personal Planner* shows these abbreviations in action:

```
(defmacro define-command
    (command-names user-documentation &body body)
  (let ((command-names (if (listp command-names)
                           command-names
                           (list command-names))))
    `(add-command ',command-names
                  ,user-documentation
                  #'(lambda ()
                      (format t "~&~A~%" ,user-documentation)
                      ,@body))))
```

The first two abbreviations occur so frequently that the expressions they replace have become virtually obsolete. Their popularity certainly is tied to their convenience, and their meaning is well understood by all Lisp programmers.

The third abbreviation helps to simplify macro syntax by allowing a more natural means of expressing the body of a macro. The *backquote syntax* provides a mechanism for specifying which parts of the macro will be evaluated and which parts are static. With this in mind, you can adopt a simple technique for writing macro bodies:

- First write out the expression that should result when the body of the macro is executed, and place the backquote character in front of that list.
- Then determine which parts of that expression are static and which parts are variables whose value will be determined through evaluating arguments supplied by the construct invoking the macro; place a comma before all those expressions that need evaluation.
- If the evaluation involves a list, check the nesting level that should result during evaluation; if the list should be seamlessly incorporated into the expression at the current nesting level, add the at-sign character between the comma and the list variable.
- Your macro body should now be close to its completed form.

See Section 4.2.5 for a discussion of the proper use of macros.

3.4.2 Some Common Lisp Idioms

A common technique in Lisp involves using a destructive function in a locally controlled situation, as in the following example of resources. A *resource* is a pool of objects from which you can allocate Lisp objects. When you are finished using an object, you can return it to the pool for subsequent allocation. If the resource is empty when you first try to allocate something from it, a fresh object is created; otherwise, the object is removed from the resource. The net effect is that, in time, there will almost always be some objects available in the resource, and fresh objects do not need to be created.

```
(defvar *freelist* '() "A list of free cons cells")

;;; This function allocates a list of LENGTH cons cells.
(defun allocate-list (n)
  "Return a list of N cons cells, allocating as many as possible
   from *freelist*, otherwise consing them afresh."
```

```
(if (not (plusp n))
    ;; No cons cells are requested.
    nil
    ;; At least one cons cell is requested.
    (let* ((result (or *freelist* (cons nil nil)))
           (last-cons result))
      (dotimes (i (1- n))
        (if (null (cdr last-cons))
            ;; There aren't enough CONS cells in the freelist, so
            ;; allocate one.  Note the destructive SETF operation
            ;; extends the result list.
            (setf (cdr last-cons) (cons nil nil))
            ;; Else we're using a cons cell from the freelist.  To
            ;; guarantee that the user will be given a list of NILs,
            ;; set the CAR of this cons to NIL to clear out old junk.
            (setf (car last-cons) nil)
            )
        (setq last-cons (cdr last-cons)))
      ;; The shortened freelist starts at the cdr of the last cons.
      (setq *freelist* (cdr last-cons))
      ;; Smash the cdr of the last cons to NIL to terminate the result.
      (setf (cdr last-cons) nil)
      result)))

;;; This function returns LIST to the pool of free cons cells.
(defun free-list (list)
  "Return LIST to the pool of conses."
  (setq *freelist* (nconc list *freelist*)))

;;; This macro executes BODY with LISTVAR bound to a list of LENGTH
;;; cons cells.  This is the safest way to use the resource;
;;; it calls ALLOCATE-LIST before executing the body and
;;; guarantees that the list is always returned to the freelist
;;; after the execution of the body completes.
(defmacro with-list ((listvar length) &body body)
  `(let ((,listvar (allocate-list ,length)))
     (unwind-protect
          (progn ,@body)
       (free-list ,listvar))))
```

By using destructive operations, we can maintain the pool of resources while keeping storage requirements to a minimum. We only create new objects if there are not enough free cons cells available. Since any cons cells that we create are spliced into the free list when we are done with them, the list continues to grow. Thus, the situations that require more cons cells than are available decrease the more we make use of the pool.

Note that the use of cons cells as a resource is completely contrived. Objects kept in a resource are usually those difficult to create and allocate at run-time, like bitmaps.

The following example uses destructive operations idiomatically as in the preceding example, but it demonstrates another common idiom—building up a list backwards and reversing it at the end to preserve the chronology of the input:

```
(defun find-events (starting-time ending-time)
  ;; Return a list of events scheduled for the interval between
  ;; starting-time and ending-time, in chronological order.
  (let ((events '()))
    (map-events starting-time ending-time
                #'(lambda (event) (push event events)))
    (nreverse events)))
```

This destructive modification is locally controlled and thus appropriate and idiomatic. The variable being modified is not used in any other binding environment; it is local to the `let` expression. Since we are not interested in any intermediate values of `events` but only in the final value, we can replace its original structure with one that better suits our needs.

Steele mentions a standard idiom in which `mapcan` is a filter to destructively transform input data into the desired output data. Again, this type of destructive modification is well specified and controlled. In such a case, the resulting destructive behavior is desirable and can be more efficient than its nondestructive counterpart.

The special form `progn` has an idiomatic relationship with the conditional construct `if`. If you have an either-or conditional expression that uses `if` and you want to execute more than one expression as a result of the condition being satisfied, you can wrap a `progn` construct around the group of expressions.

```
(defun page-thumber (chapters total-pages)
  ;; If there are no more chapters, print the total to the
  ;; stream and to the terminal.
  (if (end chapters)
      (progn (format *out-stream* "~%~A has ~S total pages.~%"
                     *title* total-pages)
             (format t "~%~A has ~S total pages.~%"
                     *title* total-pages))
      ;; Otherwise, prompt for the current number of pages in the
      ;; chapters.
      (page-thumber (cdr chapters)
                    (+ total-pages (prompter (car chapters))))))
```

When you are not interested in having a function return values, you can include in the function definition a call to the function `values` with no arguments to idiomatically express that no values are returned. The function `values` is also the standard way of

restricting a function that could possibly return multiple values to returning a single value; the single return value is the argument to `values` in the function definition.

3.4.3 Personal Idioms

Of course, you can create your own idiomatic expressions. Successful idioms conform to the following guidelines:

- Any idiom must be easier to understand than the alternative it replaces.
- An idiom should be easier to use than its alternative.
- The contexts for using the idiom should be well specified.
- The idiom should be used consistently in those contexts.

Anyone who reads your code should be able to infer the correct interpretation for the idiom. Avoid trying to make idiomatic expressions so clever that they only make sense to you; obscure idioms are not shortcuts in the long run.

In a multiprogrammer project, idiomatic consistency is especially important. The style used by all programmers should be idiomatic to the program itself instead of a patchwork of personal idioms with little stylistic balance.

Chapter 4

PUTTING THE PIECES TOGETHER

4.1 USING PROGRAM TOOLS WELL

So far we have seen that the first steps to good programming style are establishing a sound program plan and using the proper tools to carry out the plan. This chapter presents ideas for refining those well-organized, well-written first pieces of your program.

- Section 4.2, Abstraction, discusses ways of making a program more modular by writing functions and macros that layer operations, by removing dependence on the structure of data, and by using object-oriented programming techniques.
- Section 4.3, Intermodule Interfaces, presents some guidelines for establishing effective interfaces between the components of a program.
- Section 4.4, Fine-Tuning a Program, describes the iterative process of revising a program and presents examples from the *Personal Planner*.

In this chapter, the order in which we present the material is intentional. Fine-tuning is discussed last because you do not need to consider it until after you have a working version of your program. Similarly, you cannot write good interfaces to modules if your program is not modular. Of course, all these techniques depend on good choices of algorithms, data structures, and constructs to be completely successful.

4.2 ABSTRACTION

Abstraction is the technique of separating the characteristics of an object from its actual structure. Many programmers are confused by abstraction and think of it as difficult. Yet they unwittingly use abstraction all the time in everyday programming. Abstraction is an integral part of any programming language; for example, it is integral to Lisp because

Lisp objects are represented by symbols. In the expression `(car mylist)`, the symbol `car` represents an operation performed by the function `car` on a given list. We do not have to know how `car` is implemented as long as it behaves as we expect. Without the luxury of abstract symbol names, we would have to replace the symbol `car` with the definition of the function `car` in order to retrieve the first element of `mylist`.

We often use symbols in conversation to represent abstract concepts, such as freedom and beauty; to represent general groups of objects, such as human beings; or to capture a general class of actions, such as driving a car. The symbols serve to classify the object or action—to provide us with information that the object or action embodies or that it shares with other such objects or actions. For example, a person described as an "adult American male Caucasian" shares some qualities with an "adult French female Caucasian," that is, those qualities related to age and race. He also shares qualities with a person described as an "adolescent American male Caucasian," that is, those qualities related to nationality, sex, and race. The act of writing a novel has commonality with the act of writing a play, a poem, or a song.

In a similar manner, Lisp symbols represent the basic means of introducing abstract programming concepts and constructs. The full range of abstraction lies not in the symbols themselves but in the objects, values, and definitions that are associated with the names of the symbols.

Abstraction in Lisp relies on the following basic building blocks:

- The ability to attach a name to an object or an expression through binding or assignment is the simplest form of abstraction in Lisp. Naming serves as a handle that allows you to more easily manipulate objects and data. It allows you to build more complex abstractions.
- Data abstractions allow you to capture the behavior of the data used by a program rather than concentrating on the details of the structure of the data.
- A function definition is a common means of abstraction that suppresses the details of a task from the user and that can be built upon to define other abstractions.
- Macros allow you to write expressions that do not obey the usual rules for evaluation.
- Objects from the Common Lisp Object System are abstractions that capture some behavior and that are manipulated through a set of supported operations, or *protocols*.

You can use these constructs to control the depth and extent of the abstractions that your program employs. In general, the types of abstractions that you can create with these constructs fall into a few broad categories:

- Those that hide the details of an operation from the users of it.
- Those that form an interface to separate the behavior of the data from its structure.
- Those that organize data into a hierarchy.
- Those that collapse the operations performed by several similar operators into a single, more general operation.
- Those that set up the contract of a protocol.

The following sections explore how the building blocks of abstraction work in these contexts.

4.2.1 The Simplest Abstractions

The most common programming abstractions involve associating names with data as a convenient and efficient means of using those data in repeated operations. For example, consider the following function, which uses the sum of its two arguments in several operations:

```
> (defun manipulate-args-directly (x y)
    (* (+ 5 (+ x y)) (- 100 (+ x y)) (+ x y)))
MANIPULATE-ARGS-DIRECTLY
> (manipulate-args-directly 3 5)
9568
```

The variables `x` and `y` are added together every time their sum is needed as an operand. The following example binds a temporary variable to the sum of the two arguments; the sum is only computed once. Thereafter, the variable `sum` serves as the operand in the appropriate places:

```
> (defun manipulate-args-abstractly (x y)
    (let ((sum (+ x y)))
      (* (+ 5 sum) (- 100 sum) sum)))
MANIPULATE-ARGS-ABSTRACTLY
> (manipulate-args-abstractly 3 5)
9568
```

This simple abstraction not only makes the second definition easier to read than the first but also reduces the number of calls to the function + from three to one.

Repetitive code of any form is generally a good candidate for an abstraction. In some cases, the repetition might not be as blatant as the examples shown here; that is, there might be slight differences in each instance of the code. In these cases, look for the common threads and see if they can be abstracted. Then consider whether rewriting to take

advantage of this commonality simplifies your code or makes it more difficult to follow. As a general rule of thumb, the abstraction is worthwhile if it is simpler than the code you started with.

Another common abstraction involves giving names to values that are important to your program. For example, suppose you are writing a cost analysis program for the Postal Service. Among the figures that are important to the calculations and projections of the program is the cost of a first-class stamp. In every equation involving stamps, you might use the integer 25 to represent the price of a stamp. However, each time the Postal Service changes its rates, you must locate every instance of 25, see if it pertains to stamps, and then increase it. A better solution involves defining a global parameter named `first-class-stamp-price` and then using that parameter in stamp expense calculations. Then, when the price of a first-class stamp goes up to 28 cents, you would only have to change the single definition of this parameter in order to make the new value pervasive throughout the program.

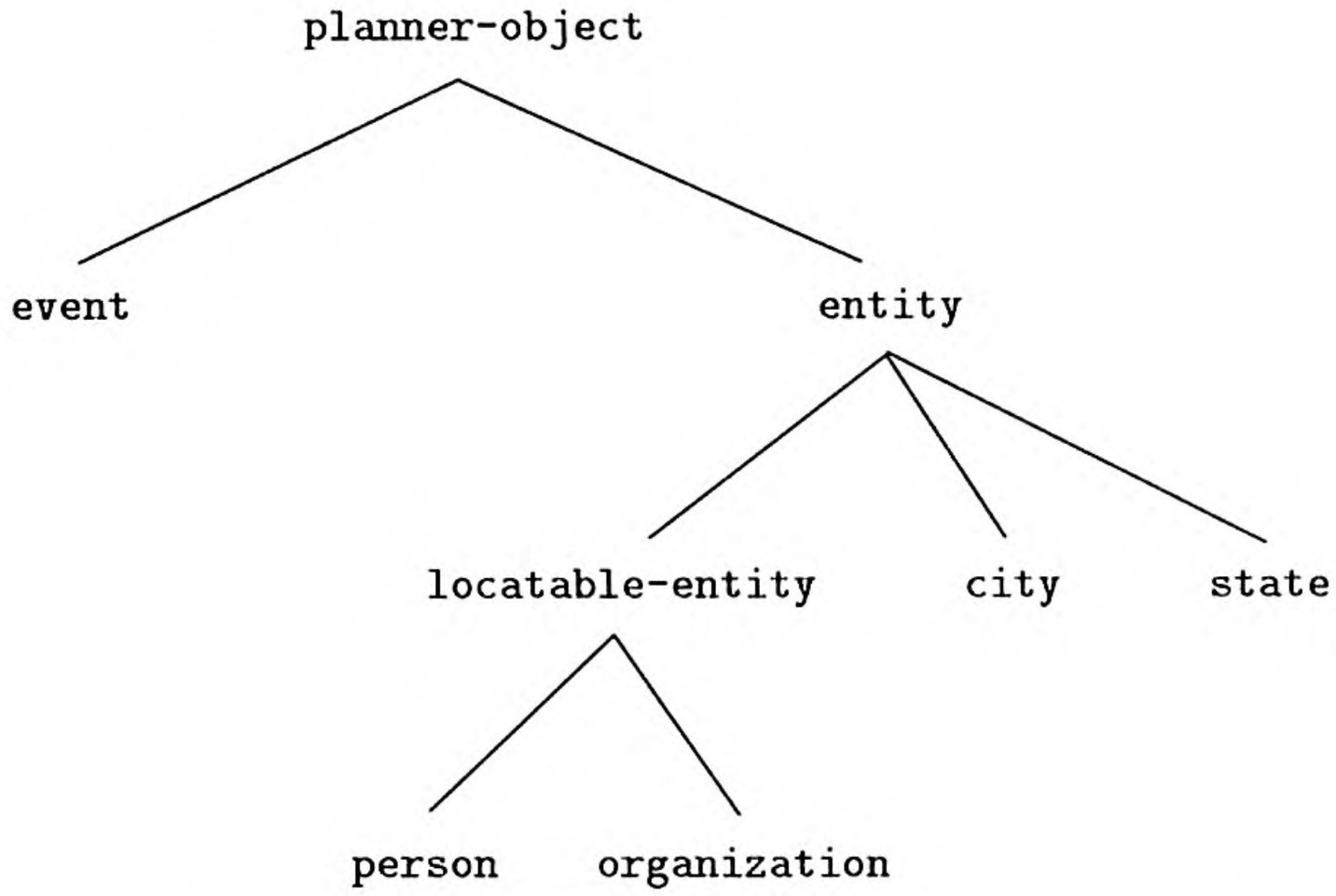
This abstraction technique is referred to as *information hiding*. That is, the abstraction hides details that the user does not need to know to execute the program correctly.

You will see the principles illustrated in these two simple abstractions used again and again in well-written programs.

4.2.2 Data Hierarchies

Like symbols, data structures are abstract representations of information. Thus, data structures are among those abstractions that arise in the most simple of everyday programs. One way to build upon these abstractions is to arrange data structures in a hierarchy. The top level of the hierarchy contains the most general description of the data structures; any data structure in the hierarchy must fit this description. Operators that manipulate top-level data structures work on all the other data structures in the hierarchy. The lower you descend in the hierarchy, the more you lose this generality. This arrangement allows you the greatest economy of operations and is thus a very useful abstraction.

The *Personal Planner* organizes its objects into a data structure hierarchy whose upper levels look roughly like those in the figure on page 69. Since we cannot know in advance all the attributes that any object might need, we can use one slot of every object to hold a property list. Using a property list allows us to add properties when and if they are required. Any object's property list can be arbitrarily long and can contain any Lisp data type; thus, objects of the same class can have distinctive properties as the contents of their property lists.



To get such a property list slot, we define a superclass of all entities and events, which we call a planner object in honor of the *Personal Planner*; the planner object is designed to inherit a property list mixin:

```
;;; Events and entities both inherit from the PLANNER-OBJECT, which
;;; has the property-list-mixin.
(defclass planner-object (property-list-mixin)
  ())

(defclass property-list-mixin ()
  ((property-list :type list
                  :accessor property-list
                  :initform '())))
```

The planner object is a *basic class* that adds a layer of abstraction. It provides a basic functionality that all objects emanating from it share through inheritance. In this case, the other objects can inherit a property list from the planner object. The planner object itself is not instantiated.

A *mixin* is a special class that defines a particular feature of an object; the class definition of a mixin is incomplete, so mixins are never directly instantiated. Through multiple inheritance of mixins, the definitions of other objects can grow incrementally.

All the other objects in the *Personal Planner* are built up from the planner object, so they too inherit the property list mixin. With this type of abstraction, we can add any other features to our objects automatically by expanding the definition of the planner object.

The planner object is a superclass of all other objects in the program. The objects that inherit from the planner object are its subclasses. The subclasses in turn are superclasses

to other objects, which will also inherit from the planner object. Thus, all objects in the *Personal Planner* directly or indirectly inherit a property list on which we can store any type of information. In addition, each object inherits the following methods for accessing and updating its property list:

```
(defmethod get-property ((object property-list-mixin) property)
  (getf (property-list object) property))

(defmethod set-property ((object property-list-mixin) property value)
  (setf (getf (property-list object) property) value))

(defmethod remove-property ((object property-list-mixin) property)
  (remf (property-list object) property))
```

There are two direct descendants of the planner object—entities and events. Each of these descendants has specific qualities and functions that do not generalize upwards to the planner object but that are inherited by their own descendants. For example, entities form another data hierarchy within the planner object hierarchy. Entities are also a type of basic class, since they are not directly instantiated. Any object that inherits from the entity class has a name that is a string, a list of nicknames, which can be `'()`, and accessors for both the name and the nicknames; it also inherits a property list from the planner object.

```
;;; Entities include locations, people, organizations, i.e.,
;;; everything except events.  Every entity has a name and
;;; a list of nicknames. Every entity is stored in the entity
;;; table under all its names.
(defclass entity (planner-object)
  ((name :type string :accessor entity-name)
   (nicknames :type list :initform '() :accessor entity-nicknames)))
```

A print method is associated with entity objects, which the standard generic function `print-object` invokes to print any named object and its type. The Lisp printer calls this method for any event object that is not shadowed by a more specific print method.

```
(defmethod print-object ((entity entity) stream)
  (format stream
          "#<~:(~A~) ~S>"
          (class-name (class-of entity))
          (entity-name entity)))
```

A subclass of entity can have its own specific operations that do not generalize upwards. For example, person objects have qualities that distinguish them from other entities. One of those qualities is a title, which describes the person's job title. We might want to write a print method for a person object that includes this title. If we did so, the Lisp printer

would use this specific method to print out any person object. Any object that inherited from person objects would thus be printed with this method instead of with the entity print method.

In addition to allowing for a large degree of shared functionality, data type hierarchies provide a clear picture of the data and how they interrelate. When you add a new data type, you must first consider how it corresponds to the existing types—how it fits into the hierarchical structure. In doing so, you begin to define the functionality for the new data type; you know what type of behavior it can inherit and what it must provide on its own. In this way, the hierarchy itself provides information about the data that might otherwise be obscured in a different form of organization.

4.2.3 Data Abstraction

Another common abstraction involving data is called data abstraction. *Data abstraction* is a method of defining data in terms of a natural collection of access and update operations, thereby minimizing the importance of the underlying data representation. A good data abstraction allows the programmer freedom to modify underlying data structures without adversely affecting other parts of the program. By the same token, data abstraction allows the maintainer of the program flexibility in updating data representations to reflect changes in the types and amounts of data being processed. The differences in the choice of data structure are limited to the representation; they are not functional differences.

For example, in the initial description of the programmatic interface to the *Personal Planner*, we defined a phone number in terms of its predicate and accessors:

```
(PHONE-NUMBER-P object)
(PHONE-NUMBER-AREA-CODE phone-number)
(PHONE-NUMBER-LOCAL phone-number)
(PHONE-NUMBER-EXTENSION phone-number)
```

These are the operations that we need to perform on phone numbers, so any representation of a phone number data type must allow for an area code, a local listing, and an extension. The representation can take any form that includes this information. Other functions that require phone number information do not interact directly with phone number objects but use these accessors to obtain the information. If the data representation that we first used in the program changes during the development of the program, we need only revise the definitions of these accessors. The functions that use information provided by the accessors do not need to be modified. This abstraction technique is called establishing *abstraction barriers*. You can ignore to a great degree the choice of data structure by constructing abstraction barriers between different levels of abstraction.

We have already described another type of abstraction barrier in the *Personal Planner*. As shown in Chapter 2, our initial description included both a programmatic interface and a user interface. The user interface is layered on top of the programmatic interface.

Abstraction barriers are built into CLOS. Though each object includes slots that specify which characteristics are possible for the object, it is the methods for the object that actually define its behavior. Any changes that you make to the underlying structure of an object are generally isolated from the rest of the program by its methods, since they provide the most common means of access to the object and its slots.

You can create a new object in two ways—structurally, by having the object inherit slots of a new class, or functionally, by defining new methods for the object to change its behavior. You can combine methods for an even greater variety and range of behavior. See Section 4.2.6 for more information about abstraction in CLOS.

4.2.4 Functions

Functions, like symbols, are abstractions in themselves, abstractions that are an indispensable part of programming. A function definition groups together an abstract set of operations. These operations are performed whenever there is a valid call to the function. In some cases, the function definitions apply only to a narrow set of circumstances, while in other cases, the function might operate on many types of data.

It is important to keep function definitions short and simple. The first step toward meeting these goals is to construct every function to perform a single, well-defined task. In other words, every function definition captures a distinct idea. The functions defined for your program correspond to the pieces of the problem identified in the early planning stage (see Chapter 2). Functions should be put together in obvious ways so that the algorithm and the means of expression you use make the solution obvious.

A user-callable function should check the types of its arguments; then, if an argument is not valid, it is never processed. In this way, lower-level functions can assume that they will receive arguments of the correct type. This guideline is primarily of interest to Lisp implementors and to writers of library routines and user-callable applications.

When writing a function definition, mentally step through it to test the results it returns; you can then test how it interacts with other functions within the Lisp environment. By incrementally testing definitions as you program, you can catch errors early, before they have caused your program to stray far from the intended path. If you find that a part of the solution does not work as expected, you can localize your changes to the function definition that is in error. Similarly, by isolating tasks into simple, well-defined units, you can easily add new functionality.

When possible, avoid writing function definitions that are longer than a screenful of text. This guideline is intended to be vague, since screen requirements change from site to site. However, anyone reading a function definition should be able to digest it wholly without having to scroll the buffer in the available window. Such size restrictions contribute to the overall readability of your program from a strictly aesthetic viewpoint, but by necessity they also support the goal established earlier of restricting the number of tasks in any given function definition. If it takes 100 to 200 lines to define a function, the function might be trying to perform more operations than it should.

4.2.4.1 SUBROUTINING

The most common functional abstraction in Lisp could be called *subroutining*; that is, replacing an operation common to several function definitions with a call to a new function that performs the operation for all interested parties. For example, suppose you have a program that uses several different vectors to store Lisp objects. As a debugging tool, in each vector's update operation you include a print operation that prints the contents of the vector whenever a certain flag is set. However, since all the print operations involve vectors, you soon realize that it would be easy to write a general print function that takes a vector as an argument and then prints the contents. By pulling out the separate operations into a common utility, you can make your program more modular and reduce the number of tasks that each update operation performs. You might later generalize this print function to write out any array type, thereby increasing the utility of the abstraction.

4.2.4.2 ANONYMOUS FUNCTIONS

In addition to named functions, Common Lisp allows you to write anonymous functions in the form of lambda expressions. All function and macro definitions contain an implicit lambda expression; that is, function and macro definitions expand into lambda expressions associated with a name. The values of the lambda list parameters are obtained from outside of the definitions and are manipulated by the lambda body. You can also embed lambda expressions within a function or macro definition when the execution of the enclosing function or macro requires some auxiliary action but a separate, named function is unwarranted. Such situations occur frequently in Common Lisp because certain Lisp functions, such as the mapping functions, `funcall`, and `apply`, take function objects as arguments.

Consider the following example from the *Personal Planner*:

```
(defun map-events (starting-time ending-time fun)
  ;; Apply FUN to every event between starting-time and ending-time
  ;; in chronological order.
  (map-event-tree starting-time ending-time *event-table* fun))

(defun find-events (starting-time ending-time)
  ;; Return a list of events scheduled for the interval between
  ;; starting-time and ending-time, in chronological order.
  (let ((events '()))
    (map-events starting-time ending-time
                #'(lambda (event) (push event events)))
    (nreverse events)))
```

The function `map-events` takes a function argument that operates on all the known events within a given interval. The single task of `find-events` is to build a list, and it uses a lambda expression to accomplish this task. In this instance, the lambda expression is embedded within the function `find-events`, and the resulting definition is easier to understand than if an additional function call had been made. Whenever `map-events` finds an event within the given interval, it passes the event directly to the lambda expression, which pushes it onto the list. The definition is short, simple, and to the point.

4.2.4.3 GENERIC FUNCTIONS

If you find that several functions perform similar operations on different types of data, consider whether the definitions can be collapsed into one; that is, replace repetitive code with a call to a general function that handles the task abstractly. If several functions perform the same operation on different types of data, think about what would be required to construct a *generic function* to perform the task. Though CLOS defines a generic function data type, for the purposes of this discussion we consider a generic function to be any Common Lisp function that provides a single interface to a set of different operations depending upon the type associated with their arguments. (In fact, CLOS is inspired by the type generality of regular Common Lisp functions, such as +.)

Some of the most commonly used functions in Common Lisp are generic functions. For example, the arithmetic functions work with all numeric data types—`integer`, `fixnum`, `bignum`, `float`, `rational`, and `complex`:

```
> (+ 1 3.5)
4.5
> (/ (/ 4 3) 0.6)
2.2222222222222223
> (* #c(100 45) #c(0 45))
#C(-2025 4500)
```

The function `inspect` can accept any Lisp object. Functions in the file system interface that require a pathname argument can accept any of the following objects:

- A pathname
- A string that can be coerced to a pathname
- A stream that is open to a file

The Common Lisp sequence functions also provide good examples of generic functions. Consider the following definition of a function that is similar to the Common Lisp function `reverse`:

```
(defun my-reverse (sequence)
  "Returns a new sequence containing the same elements but in
   reverse order."
  (etypecase sequence
    (list (list-reverse sequence))
    (vector (vector-reverse sequence))))
```

This function takes a *sequence* argument, which can be a list, a string, or a vector. The single task performed by this function is to dispatch on the type of its argument by using the Common Lisp macro `etypecase`; all other tasks are performed by functions invoked within the body of the `etypecase`. If the argument is a list, the function `list-reverse` is called. If the argument is a vector, `vector-reverse` is called; since strings are vectors, they are covered by this case. If the argument is neither a list or a vector, `etypecase` signals an error:

```
> (my-reverse '(a b c))
(C B A)
> (my-reverse 1)
>>Error: The value of SEQUENCE, 1, should be a VECTOR or a LIST
```

With this setup, the functions `list-reverse` and `vector-reverse` do not have to check the types of their arguments, since this task has already been done. Their task is simply to store the elements of the argument in reverse order in a new sequence and to return that sequence to `my-reverse`.

CLOS allows you to define a generic function that takes arguments of any type; in this way, it lets you define an operation that applies to several classes of objects at once. In CLOS, a generic function serves as an interface between an object and the methods that perform operations on instances of that object. The generic function simply dispatches to the correct method for the type of object being operated on. By defining methods for an object, the interface of the generic function is extended to cover a new class. There are

many predefined generic functions in CLOS, such as `change-class`, `make-instance`, and `shared-initialize`. These generic functions are all standard generic functions; that is, they use standard method combination, which allows primary methods to be combined with `:before`, `:after`, and `:around` methods. See Haskell and Miller for a discussion of generic functions and of standard method combination.

One of the standard generic functions in CLOS is `print-object`, which prints an object to a stream. This generic function recognizes any user-defined print methods that describe how a particular instance of an object is printed. Whenever you want to write out the printed representation of the object, the generic function invokes the correct print method for that object. Every print method must abide by certain rules concerning the standard printing flags of Common Lisp, such as `*print-length*`, but thereafter the similarities can end. Consider the following print methods from the *Personal Planner*:

```
;;; This method prints an entity.
(defmethod print-object ((entity entity) stream)
  (format stream
          "#<~:(~A~) ~S>"
          (class-name (class-of entity))
          (entity-name entity)))

;;; This method prints an event.
(defmethod print-object ((object event) stream)
  (with-slots (starting-time location duration) object
    (format stream "#<~:(~A~) at " (class-name (class-of object)))
    (write-date-and-time starting-time stream
                         :timezone
                         (location-time-zone starting-location))
    (when (> duration 0)
      (write-string " for " stream)
      (write-time-interval duration stream nil))
    (write-char #\> stream)))

;;; This method prints an address.
(defmethod print-object ((address address) stream)
  (with-slots (street-or-po-box city zip-code)
    address
    (format stream "#<Address ~A, ~A, ~A ~5,'0D>"
            street-or-po-box
            (entity-name city)
            (state-abbreviation (city-state city))
            zip-code)))
```

The printed representation of an instance of the entity class consists simply of the name of its class followed by a colon and the entity name. The printed representation of an event instance is more complicated. It contains the name of the event class, the starting time of the event, and in some cases the duration of the event. The printed representation of an address instance is the most complex. It begins with an identifier that tags it as an address. The identifier is followed by a street, city, state, five spaces, and a zip code. The print methods for instances of these objects are clearly disjoint, yet the user does not have to be aware of the differences, since a single generic function handles the printing of all types of instances.

4.2.5 Macros

The previous sections emphasized that abstraction is easy and that you use it all the time in everyday programming. This generalization does not completely apply to macros, and as a result, macros are sometimes misused.

A macro performs source-to-source transformations; that is, it translates one Lisp expression into another Lisp expression that will be evaluated. Writing a macro is like writing a *metaprogram*, a program that defines a program. Unlike arguments to functions, a macro's arguments are not evaluated. Most macros are written using backquote notation to allow the syntax of the macro body to more closely resemble that of a function definition (backquote notation is described in Section 3.4.1). In the discussion that follows, we assume that macros are written using backquote notation, and we follow this practice in our examples.

When you use backquote notation within the macro body, you must explicitly designate which forms should be evaluated. The result of the evaluation is substituted in place into a copy of the template. This process of evaluating certain portions of the template in place and producing a form that can be evaluated is called *macro expansion*.

As a general rule, we suggest you use macros only in cases where a normal function evaluation would not produce the desired result, that is, for syntax. Note that a macro should expand into something simple; it should use the minimum amount of syntax necessary to transform its body into an evaluable expression. Using macros is appropriate in three common situations:

- For defining forms
- To set up a dynamic context for evaluating certain forms
- For control structures

You are probably familiar with several examples of macros that define forms, the most common of which is `defun`—the form for defining functions. Such a form needs an interface similar to that of a normal function call yet must have the power to transform itself into a form that can get the desired results from the arguments. The macro definition shields the user from unnecessary details; the user does not need to know how the macro works but only how the macro expansion function will operate on the specified arguments.

For example, suppose you needed a way to define numeric constants over a given range. You would like to specify the low and high ends of a range and a list of names and would want the results from those data to be a series of numeric constants of the given names defined in that range. The following macro performs such an operation:

```
(defmacro defrange (starting-value next-value-past-range names)
  (if (null names)
    ;; The degenerate case.
    '()
    (let ((first-constant (first names)))
      `(progn
         (defconstant ,first-constant ,starting-value)
         ,@(loop for i from 1
                 for name in (rest names)
                 collect
                 `(defconstant ,name (+ ,first-constant ,i)))
         (unless (< ,(first (last names))
                    ,next-value-past-range)
           (warn "~S exceeds specified limit" 'defrange))))))
```

When invoked, this macro would expand roughly into the following expression:

```
> (defrange 0 4 (foo bar baz mumble))

          ==>

(progn (DEFCONSTANT FOO 0)
       (DEFCONSTANT BAR (+ FOO 1))
       (DEFCONSTANT BAZ (+ FOO 2))
       (DEFCONSTANT MUMBLE (+ FOO 3))
       (UNLESS (< MUMBLE 4)
               (WARN "~s exceeds specified limit"
                     'DEFRANGE)))
```

The macro controls the syntax of the range definition and expands into a `progn` in which a series of Lisp constants are defined. In such a situation, a macro is not only the most appropriate abstraction for the job but also the most effective.

The second situation where macros are appropriate arises when you want to control the dynamic context of the Lisp environment during the execution of certain forms and to restore the environment to its original state when the execution of the forms is complete. In other words, for a brief period of time, you need to control the execution environment of the macro. For example, suppose you wanted to ensure that a certain portion of Lisp code could not be interrupted while it was executing. You could write a macro to prevent any interrupts from being seen while the body of the code, which is spliced into the macro in place, executes. When every form in the body has completed, the macro relinquishes control of the evaluation environment.

```
(defmacro with-interrupts-deferred (&body forms)
  `(unwind-protect
       (let ((*defer-interrupts* t))        ;Turn off interrupts.
         ,@forms)
     (unless *defer-interrupts*
       (check-deferred-interrupts))))
```

A common component of these types of macro definitions is the Common Lisp special form `unwind-protect`, which ensures that certain cleanup forms are executed before the macro exits. Note that `let` bindings of special variables contain an implicit `unwind-protect`. If the `let` form exits in any way, the special variable is restored to the value it had before the `let` binding.

The third situation—control structures—does not arise very often. You can generally get by with using the rich vocabulary of control constructs provided by Common Lisp. However, if your program repeatedly manipulates some data structures that are not effectively handled by any Common Lisp control construct, you might consider writing a specialized control macro. For example, suppose your program needs to repeatedly iterate over the contents of a vector. You might write a specialized form of `do`, as the following example demonstrates:

```
;;; DOVECTOR evaluates the form VECTORFORM, which should produce a
;;; vector.  It then executes the body once for each element of the
;;; vector in turn, with the variable VAR bound to the element.
;;; RESULTFORM is evaluated, and the result is the value returned
;;; by the form.  If RESULTFORM is omitted, DOVECTOR returns NIL.
(defmacro dovector ((var vectorform &optional (resultform nil))
                    &body body
                    &environment environment)
  "Like DOLIST, but works on vectors and one-dimensional arrays."
  (check-type var symbol)
```

```
;; Generate unique names for temporary variables.
(let ((vectorvar (gensym "VECTOR"))
      (lengthvar (gensym "VECTOR-LENGTH"))
      (indexvar (gensym "VECTOR-INDEX")))
  (multiple-value-bind (bare-body declarations)
      ;; PARSE-BODY extracts any declarations in the body.
      (parse-body body environment nil)
    `(let* ((,vectorvar (values ,vectorform))  ;One evaluation only.
            (,lengthvar (length ,vectorvar)))
       (do (,var                                ;SETQ'd below.
            (,indexvar 0 (1+ ,indexvar)))
           ((= ,indexvar ,lengthvar) ,resultform)
         (declare (fixnum ,indexvar))
         ,@declarations
         (setq ,var (aref ,vectorvar ,indexvar))
         ,@bare-body
         )))))
```

The macro dovector expands into a do form:

```
> (pprint (macroexpand-1 '(dovector (item vector) (print item))))

(LET* ((VECTOR1 (VALUES VECTOR))
       (VECTOR-LENGTH2 (LENGTH VECTOR1)))
  (DO (ITEM
       (VECTOR-INDEX3 0 (1+ VECTOR-INDEX3)))
      ((= VECTOR-INDEX3 VECTOR-LENGTH2)
       NIL)
    (DECLARE (FIXNUM VECTOR-INDEX3))
    (SETQ ITEM (AREF VECTOR1 VECTOR-INDEX3))
    (PRINT ITEM)))
```

It is important to keep in mind that the number of appropriate occasions for using macros is limited. Consider, for example, how many macros are documented by Steele. Out of 775 constructs in the first edition of his book, only 59 are macros—less than 10 percent. Forty-four of these Common Lisp macros can be grouped into one of the three categories mentioned previously. By looking over the following lists, you can get a general feeling for the types of operations that fall into these groupings.

These macros are used for defining forms and objects:

defconstant	define-modify-macro
define-setf-method	defmacro
defparameter	defsetf
defstruct	deftype
defun	defvar

These macros are used to control the dynamic context of evaluation:

```
step                          time
trace                         with-input-from-string
with-open-file                with-open-stream
with-output-to-string
```

These macros are used as control structures:

```
and                           assert
case                          ccase
check-type                    cond
ctypecase                     do
do*                           do-all-symbols
do-external-symbols           do-symbols
dolist                        dotimes
ecase                         etypecase
locally                       loop
multiple-value-bind           multiple-value-list
multiple-value-setq           or
return                        typecase
unless                        untrace
when
```

(The macros `setf`, `psetf`, `psetq`, `incf`, `decf`, `remf`, `rotatef`, and `shiftf` are special cases. They add a new feature to the language—generalized variable assignment. The macros `pop`, `prog`, `progn`, `prog1`, `prog2`, `push`, and `pushnew` provide generalized variable access. Using macros to add new features to Lisp is generally unnecessary.)

Macros are powerful abstractions because they allow you to specify completely how the syntax is evaluated. When macros are processed, the result of their expansion is substituted in line. As a result, macros do not require the overhead of a function call and thus run faster than functions. However, they are more difficult than functions to read, write, and debug. You must consider whether the speed and power you gain with judicious use of macros outweighs the clarity and maintainability that the program would have without them.

Note that whenever you redefine a macro, you must recompile every function that uses it for the new definition to take effect. If you fail to recompile all uses of the new definition, both definitions of the macro might be active in the same environment. This problem is difficult to detect and often has deleterious results.

4.2.5.1 MISUSE OF MACROS

Probably the most frequent misuse of macros is for in-line coding, with the rationale that using a macro saves the expense of a function call and will be faster. However, there is a more direct method for having functions open coded—`inline` compiler declarations.

For example, consider the following macro, which writes a character to the standard output device. The macro expands into a call to the macro `dotimes`, which in turn expands into a call to `do`:

```
;;; Write n copies of a character.
> (defmacro format-write-nchars (char n)
  `(dotimes (i ,n) (write-char ,char *standard-output*)))
FORMAT-WRITE-NCHARS
> (compile *)
FORMAT-WRITE-NCHARS
> (pprint (macroexpand-1 '(format-write-nchars #\m 10)))
(DOTIMES (I 10)
         (WRITE-CHAR #\m *STANDARD-OUTPUT*))
> (pprint (macroexpand-1 '(dotimes (i 10)
                                   (write-char #\m
                                               *standard-output*))))

(DO ((#:G61 10)
     (I 0 (1+ I)))
    ((>= I #:G61)
     NIL)
  (DECLARE (FIXNUM I #:G61)) (WRITE-CHAR #\m
                                         *STANDARD-OUTPUT*))
```

Using a macro in such situations is overkill; in this case, the abstraction makes the code more complex than is necessary. The following proclaimed function produces the equivalent behavior in a straightforward fashion:

```
(proclaim '(inline format-write-nchars))

(defun format-write-nchars (char n)
  (dotimes (i n) (write-char char *standard-output*)))
```

There are many reasons why it is better to use functions than macros:

- With functions, it is clear that all the arguments are evaluated.
- You can use functions as functional arguments.
- Functions are easier than macros to read and write.

- It is superfluous to write a metaprogram when a program alone is sufficient.

Another type of macro that can generally be avoided is one that invokes other macros or functions on the fly by building symbol names dynamically. Such constructions are usually difficult for the human reader of the program to follow, and they also confound Lisp editors and code walkers (code walkers are tools for locating symbols in a program). Thus, they are difficult to debug and maintain. If possible, redefine the problem in such a way that you do not need to build the names dynamically.

There can be situations in which you might use such an arrangement to good advantage, as when you must define a large class of objects with similar names; `defstruct` uses this name construction method for just this reason. However, it is interesting to note that, despite its resemblance to `defstruct`, `defclass` does not construct names on the fly. Instead, it uses a different approach that renders dynamic name construction unnecessary.

4.2.5.2 PROBLEMS TO AVOID WHEN WRITING MACROS

There are three problems to watch out for in writing macro definitions—inadvertent name capture of local variables, multiple evaluation of macro arguments, and harmful side effects within macros.

The first problem is easy to avoid. Whenever you write a macro with an `&body` argument, use the function `gensym` or `gentemp` to create a new symbol for any local variable visible within the body of the expanded form:

```
(defmacro time-loop (limit &body body)
  (let ((countvar (gensym "COUNTER")))
    `(time (dotimes (,countvar ,limit) ,@body))))
```

Otherwise, you might violate lexical scoping and have name conflicts with user-supplied variables in the `&body` argument at expansion time.

The second problem is not so easy to prevent. Because macros do not obey the usual evaluation rules, you must specify what parts of the macro body should be evaluated and any substitutions that are required. Make certain that every macro is written so that its arguments are evaluated only one time within the body. Otherwise, the macro might not only perform unnecessary work but also produce unreliable results, as the following examples show:

```
> (setq temp 5)
5

;;; This macro conses an expression and returns it.  However, this
;;; definition does not produce the value we want because the
```

```
;;; expression is evaluated one time too many.
> (defmacro set-pair (location expression)
    `(setf ,location (cons ,expression ,expression)))
SET-PAIR
> (set-pair result (incf temp))
(6 . 7)
> temp
7

;;; This version works correctly.
> (defmacro set-pair (location expression)
    (let ((value (gensym "VALUE")))
      `(let ((,value ,expression))
         (setf ,location (cons ,value ,value)))))
SET-PAIR
> (set-pair result (incf temp))
(8 . 8)
```

Step through any macro definition carefully or expand a call to it to see how many times each form within the body is evaluated. Rewrite any macro whose body forms undergo multiple evaluations.

Finally, because you might not know how many times a macro will be expanded, make certain that your macros do not contain side effects, either on global data structures or on the input form. At issue here is having side effects on the macro itself, that is, the code that runs when `macroexpand` is called. It is legitimate for the expanded form to have side effects when evaluated. For example, the macro `setf` expands into a form that causes a side effect when it is evaluated, but `setf` itself does not have any side effects; it merely conses up an expression to return.

Most macros that have global side effects at the time they are expanded can be fixed by using `eval-when`. For example, consider the following code, which is intended to ensure that a variable is proclaimed special when compiling a file that contains a `defparameter` definition:

```
(defmacro defparameter (name value &optional documentation)
  (proclaim `(special ,name))
  `(progn
     (proclaim '(special ,name))
     (setf (documentation ',name) ,documentation)
     (setq ,name ,value)))
```

However, with this definition, the following call to `macroexpand` causes the variable `a-test-parameter` to be proclaimed special at expansion time:

```
(macroexpand '(defparameter a-test-parameter 3))
```

To get the intended behavior, rewrite the macro definition as follows:

```
(defmacro defparameter (name value &optional documentation)
  `(progn
     (eval-when (eval compile load)
       (proclaim '(special ,name)))
     (setf (documentation ',name 'variable) ,documentation)
     (setq ,name ,value)))
```

Now the call to `macroexpand` has no side effects, but in this implementation, compiling a file that contains a `defparameter` form causes the variable to be proclaimed special. Thus, references in the file being compiled are compiled correctly. Notice that when the expanded form is evaluated, it now has three side effects, but they perform the needed operations:

- The call to `proclaim`
- The `setf` of `documentation`
- The `setq` of the variable

See Section 7.3.2 for a discussion of `eval-when`.

The following example demonstrates another type of unwanted side effect. The macro `constant-alist` takes alternating keys and values and returns an association list of those values:

```
(defmacro constant-alist (&rest keys-and-values)
  `',(do* ((key-tail keys-and-values next-tail)
           (value-tail (cdr key-tail) (cdr key-tail))
           (next-tail (cdr value-tail) (cdr value-tail)))
          ((null key-tail) keys-and-values)
          (setf (cdr value-tail) (car value-tail))
          (setf (car value-tail) (car key-tail))
          (setf (cdr key-tail) next-tail)
          (setf (car key-tail) value-tail)))
```

When we expand a call to this macro, it returns a quoted constant association list:

```
> (setf temp '(constant-alist a 1 b 2 c 3))
(CONSTANT-ALIST A 1 B 2 C 3)
> (macroexpand temp)
(QUOTE ((A . 1) (B . 2) (C . 3)))
T
```

However, look what happened to the original form in the process:

```
> temp
(CONSTANT-ALIST (A . 1) (B . 2) (C . 3))
```

The macro we defined actually modifies the source code of the call. This definition is very dangerous; the macro can be called more than once during evaluation, but the second time it is called, it will not return the correct form. Here is a better definition that leaves the original definition intact:

```
> (defmacro constant-alist (&rest keys-and-values)
   `',(loop for (key value) on keys-and-values by #'cddr
            collect `(,key . ,value)))
CONSTANT-ALIST
> (setf temp '(constant-alist a 1 b 2 c 3))
(CONSTANT-ALIST A 1 B 2 C 3)
> (macroexpand temp)
(QUOTE ((A . 1) (B . 2) (C . 3)))
T
> temp
(CONSTANT-ALIST A 1 B 2 C 3)
```

Note that the first definition does not require as much storage as the second definition. When you are not running interpreted code, however, it almost never matters how much storage is used when expanding a macro, since macros are expanded only once when a program is compiled and are never expanded while the compiled program is executing.

4.2.6 Object-Oriented Programming

Object-oriented programming is a new type of abstraction available in Common Lisp through CLOS. In object-oriented programming, programs consist of collections of objects that embody some behavior. Relationships among different types of objects are described through the concepts of superclasses and subclasses. A *superclass* provides characteristics and behavior that can be distributed to all of its subclasses and their subclasses through inheritance. A *subclass* can define local characteristics to complement its inherited structure. The behavior of a subclass can be further modified by combining inherited

methods with auxiliary methods or by locally overriding the behavior of inherited methods. Thus, this type of programming allows sharing of common properties and behavior for efficient use of data as well as the flexibility of layering many levels of distinct properties and actions for better program modularity. We chose object-oriented programming for the *Personal Planner* because it provided the most effective means for manipulating different types of people, places, and events.

In CLOS, all objects are members of some class, and the class forms a new data type. Associated with each object is a set of methods that manipulate instances of that type. From the user's perspective, the methods actually determine the behavioral characteristics of the instances. There are two kinds of methods: *primary methods* provide the main behavior for an instance, and *auxiliary methods* perform any setup, cleanup, or alternative operations. You can write specialized parameter lists that make methods operate only on certain instances of a class. Methods are invoked by generic functions, as described in Section 4.2.4.3. Generic functions keep objects distinct from the operations that can be performed on them. If a generic function does not exist when a method is defined, CLOS creates one from information specified in the method definition.

CLOS examines the hierarchy of superclasses and subclasses that you define in an object-oriented program to form a *class precedence list*, or CPL, for any subclass and its superclasses. The CPL determines how a class inherits properties from its superclasses by ordering the properties from most specific to most general. A property can be an object's slot or a method. The properties of a class are always more specific than properties of its superclasses. You can specify the relative precedence of two classes by including one as a superclass of the other. In the definition for a class, you can also specify the precedence of all its superclasses. If you do not specify the ordering explicitly, CLOS will determine the precedence on its own.

CLOS includes the macro `defgeneric` to define generic functions. Though you do not have to define generic functions explicitly to define methods, doing so can serve as an implicit form of documentation advertising the interface between objects and their methods. Thus, it reveals your intent to readers of your code. You can even attach a documentation string to a generic function to directly document the operation that the generic function controls. Then you can avoid having redundant descriptions of the operation for each type for which a method is defined. However, whether you use an explicit `defgeneric` form or allow CLOS to define one automatically for methods is a detail that is hidden from the user's perspective.

Because the behavior of an object is determined by its methods, there is always a clean interface between the internal representation of the object and the structure seen by the user. If the definition of the object changes throughout the lifetime of the program, the

maintainer need only revise the method definitions to make the changes visible to the user of the program. Thus, object-oriented programming supports the information-hiding concept important to most abstraction techniques.

In Section 4.2.3, we discussed the practice of defining abstraction barriers between the representation of the data and users of those data. With CLOS, you can set up another type of abstraction barrier for making instances of a class. The generic function `make-instance` creates instances of a class. However, it is a good idea to set up your own constructor functions based on `make-instance` to add a level of abstraction between the class definition for the instances and the clients that manipulate the instances. The purpose of such an abstraction barrier is the same as with accessor functions described previously—to prevent users of the data from relying on the representation of the data. Such a practice takes a lot of the pain out of updating data structures because it localizes the changes to the constructor function. In addition, a customized constructor function can be more aptly named to reflect the type of instance being created. For the *Personal Planner*, we created an abstract instance constructor with a twist (see Section 4.4). For more complete information about CLOS, see Haskell and Miller.

4.2.7 Some Points to Consider

Abstractions are meant to make your program easier to understand, both to you and to others. Thus, advertise your abstractions in comments, and explain their purpose and behavior. Keep abstractions simple, and use them clearly; do not allow an abstraction to disguise the flow of program control.

Think carefully about using macros to customize your Lisp code. The power of macros is also their weakness, for it allows you to change Lisp syntax at will. Make certain that the situation calls for a macro instead of a simpler mechanism. Do not write macros to customize your environment in such a way that it is foreign to someone else.

4.3 INTERMODULE INTERFACES

A well-written program is a collection of well-written pieces that are put together in clean, understandable ways. The glue that holds pieces of a program together is called an *intermodule interface*. We have given examples of such interfaces before, such as the interface to phone number objects described in Section 4.2.3. The interface can be written in such a way that the modules it joins require no direct knowledge of each other. In this way, the modules can be independently modified. There are a few points to remember when constructing such interfaces:

- Where possible, think of modules as distinct, and make the interfaces between them reflect the distinction.
- Ensure the integrity of the system by allowing only those modules that depend on each other to share data access.
- Use the package system to distinguish the symbols of a module (see Section 5.4.1.2).
- Keep the order of arguments consistent between modules. For example, do not confuse the user by requiring the x-coordinate first in some modules and the y-coordinate first in others. The pattern that you use for specifying arguments should be clear enough so that someone else could extend the argument list to cover new features without upsetting the pattern.
- Handle type checking at the interface. Once data have passed through the interface, the receiving functions can then assume that they are receiving correct data types. The interface detects errors in the types of the arguments it receives and performs some meaningful action in response to those errors.
- Avoid using global variables to pass values between modules. Someone reading the module can readily understand a value that has been passed in as a parameter but must rely on memory and familiarity with the program to interpret a global value.
- When the interface allows optional arguments, use `&optional` if dependence exists between some of the optional values, that is, when one of the optional values does not make sense without the others being specified. If there is no dependence and you want users to be able to specify arguments in any order and combination, use `&keyword` arguments for optional values.

4.4 FINE-TUNING A PROGRAM

The process of fine-tuning a program can best be done in small steps. Once you have a version of code that works and is in good shape, review it iteratively to see how it might be made more readable and how you might simplify its structure. Each time you review a portion of code, ask yourself a few questions:

- Would any change make the code more readable without affecting its behavior? Readability is one of the most important benefits of good programming style.
- Can you replace a complicated control structure or data structure with a simpler one? Make certain that your tools are appropriate for the jobs they perform.

- Alternatively, are the data structures appropriate for the complexity of the operations? Often simpler data structures used during the development of a program must be replaced by more powerful data structures when the program matures.
- Is there sufficient commentary to explain the intent of the operation? Is the existing commentary correct? Make certain that any revision in code involves revising the code comments as well. Do not hesitate to add commentary to someone else's code once you really understand it. Section 5.3 describes useful commentary in detail.
- Can you collapse related operations into a single, more abstract operation? Avoid redundant code.
- Can you improve local transparency by reordering your program? Section 5.4 gives some suggestions for effective physical organization of code.
- Is the code still current in light of other program developments? You might find that certain concerns and constraints of the past no longer matter. Thus, you can remove the unnecessary restrictions to make the code more understandable and efficient.

Make certain that you understand the intent of a piece of code before trying to upgrade or fix it. If you have approached the programming problem in an organized fashion and have paid attention to the style of your code along the way, any changes to the program should require an effort that is proportionate to the desired result. For general maintenance purposes, focus on changes that increase clarity. Like a classical instrument, well-worn code can become more beautiful with age.

The *Personal Planner* went through several stages of revision on the way to its final version, which appears in Appendix A. For example, when we started out, we distinguished between entities and events—entities were all the named objects, and events were all the actions involving those objects. After maintaining this distinction throughout the first draft of the program, we now realize that events can also be named. Thus, they could be subclasses of entity. This revision simplifies the program by allowing us more shared functionality. Also in the first draft, whenever we created a new entity, we stored it by name and nicknames in the hash table `*entity-table*`. The functions `add-entity`, `find-entities`, `add-nickname-to-entity`, `remove-nickname-from-entity`, `remove-entity`, `install-entity`, `delete-entity`, and `rename-entity` performed our hash table operations. The problem with this setup was that it could not be generalized beyond entities. If we had decided to have a new class called `transactions`, we would have had to create a `*transaction-table*` and all the corresponding access and update functions for transactions.

Now that we have recognized the specific nature of our initial solution to the entity database, we can make this behavior more modular by relying on CLOS inheritance. In other words, we can define a superclass that contains the storage mechanism in its slots; it is inherited by any object that we need to store.

To create this superclass, let's write a macro that defines a new version of `defclass`:

```
(defclass interned-mixin ()
  ())

;;; An interned class has a shared slot that contains the table
;;; holding all the instances of that class.  Note that subclasses
;;; should NOT use DEF-INTERNED-CLASS; the slot is shared by
;;; all the subclasses.  The class defined with DEF-INTERNED-CLASS
;;; is the root of a set of classes all of whose instances are
;;; interned together.
(defmacro def-interned-class (name superclasses slot-specifiers
                                   &rest class-options)
  `(defclass ,name (interned-mixin ,@superclasses)
     ((intern-table :allocation :class
                    :initform (make-hash-table :test #'equal))
      ,@slot-specifiers)
     ,@class-options))
```

Any class created with the macro `def-interned-class` automatically inherits the class `interned-mixin`, which is central to the protocol for `intern-instance`, described later. Each interned class contains a hash table slot for storing all instances of the class.

Now that events have become entities, we only need one interned class for the *Personal Planner*:[1]

```
;;; ENTITY is defined with DEF-INTERNED-CLASS; it is the top
;;; of a hierarchy of classes, which are kept in a table by name.
(def-interned-class entity (planner-object)
  ((name :type string :accessor entity-name :initarg :name)
   (nicknames :type (list-of string) :initform '()
              :accessor entity-nicknames)))
```

If additions to the *Personal Planner* require a database for new, nonentity objects, we can extend this storage mechanism simply by defining the new objects as interned classes.

1. We have saved our previous functional interface to the entity database in a directory of useful information. We might resurrect this mechanism if we ever want to reference a large number of events by the range of time they cover. In that case, we would need the first representation, which was ordered by time, not by name.

Note that entity also inherits from the class `planner-object`. Since it inherits from `interned-mixin` by definition, the entity class becomes our first instance of multiple inheritance.

To add data to the *Personal Planner*, the user specifies a class name such as `person`. Instead of invoking a `make-`*object* function for each type of object that the user enters, we simply call the method `intern-instance` for any specified object:

```
;;; INTERN-INSTANCE, not MAKE-INSTANCE, should be used
;;; to create objects of interned classes.  It takes initargs,
;;; just as MAKE-INSTANCE does, but will return a previously
;;; created instance if there is one in the table.
(defmethod intern-instance ((type symbol) &rest initargs)
  ;; Call the method on the prototype, just as in FIND-INSTANCE.
  (apply #'intern-instance
         (class-prototype (find-class type)) initargs))

(defmethod intern-instance ((class standard-class) &rest initargs)
  ;; Call the method on the prototype, just as in FIND-INSTANCE.
  (apply #'intern-instance
         (class-prototype class) initargs))

(defmethod intern-instance ((prototype interned-mixin)
                            &rest initargs)
  (let ((table (slot-value prototype 'intern-table))
        (key (apply #'initargs-intern-key prototype initargs)))
    (or (gethash key table)
        (setf (gethash key table)
              (apply #'make-instance
                     (class-of prototype) initargs)))))
```

The method `intern-instance` extends the behavior of `make-instance` to work like the Common Lisp function `intern` for objects of type `interned-class`. If an instance of the given class already exists in the entity hash table under the specified name, that instance is returned; if no such instance is available, `intern-instance` creates a new one by calling `make-instance` with the appropriate class information. Note that, through the use of specialized parameters, we perform these tasks with three methods named `intern-instance`. The first two methods are called for the symbol or class specified by the user, and the other is called for the class prototype of the user-specified class. (See Bobrow and Kiczales for information about `class-prototype`.)

For the *Personal Planner*, the generic function `initargs-intern-key` establishes the protocol for handling objects of type `interned-class`:

```
;;; INITARGS-INTERN-KEY takes a prototype and initargs and
;;; returns the key for those initargs.  The prototype
;;; argument is used for discrimination.  The result is
;;; used as the key in an EQUAL hash table.  Thus, it only
;;; returns values for which EQUAL makes sense, such as
;;; symbols, strings, numbers and lists of appropriate values.
;;; It does not work for vectors (other than strings) or instances.
(defgeneric initargs-intern-key (prototype &rest initargs))
```

This protocol uses the external interface defined by the methods `intern-instance` and `find-instance`. If we wanted to use a different protocol, one that does not rely on an `equal` hash table, we could set up a new protocol by defining equivalent methods for the chosen representation.

To see if this new creation and storage mechanism for entities handles our needs, we must integrate it with the rest of the program. We might even be able to improve other portions of the program in the process. For example, what if we store information about the class as a means of constructing a meaningful prompt for the user? This type of change could make our input interactions much smoother.

To explore this question, we need access to the type of information that is normally associated with a particular class. We can parse each input independently and look up the information in a table, but that requires parsing routines for every type of entity. A better solution is to let CLOS take care of the bookkeeping. We can save information about the structure of each class and distinguish between classes that the user sees and the basic classes that are never instantiated. To accomplish this behavior, we create two new types of `defclass` forms by again using macros for syntax:

```
;;; DEF-ABSTRACT-PLANNER-CLASS and DEF-CONCRETE-PLANNER-CLASS
;;; take arguments just as DEFCLASS does.  We save away some
;;; of the information for later use.

;;; DEF-PLANNER-CLASS is not used directly.  It is only
;;; defined for DEF-ABSTRACT-PLANNER-CLASS and
;;; DEF-CONCRETE-PLANNER-CLASS to expand into.
(defmacro def-planner-class
  (name abstract-p superclasses slot-specifiers
   &rest class-options)
`(progn
   (defclass ,name ,superclasses ,slot-specifiers ,@class-options)
   (save-planner-class-information
     ',abstract-p ',name ',superclasses ',slot-specifiers)))
```

```
;;; Abstract means this class will never be instantiated.
;;; It is only used as a superclass.
(defmacro def-abstract-planner-class
    (name superclasses slot-specifiers &rest class-options)
  `(def-planner-class ,name t ,superclasses
                      ,slot-specifiers ,@class-options))

;;; Concrete means this class can be instantiated.
(defmacro def-concrete-planner-class
    (name superclasses slot-specifiers &rest class-options)
  `(def-planner-class ,name nil ,superclasses ,slot-specifiers
     ,@class-options))
```

The macro `def-planner-class` creates a `defclass` form that calls the function `save-planner-class-information` to save information about all the slots defined for a class.

```
(defvar *planner-class-information* '())

(defstruct planner-class-information
  name
  abstract-p
  superclasses
  slot-specifiers)

(defun save-planner-class-information
    (name abstract-p superclasses slot-specifiers)
  (delete name *planner-class-information*
          :test #'eq
          :key #'planner-class-information-name)
  (push (make-planner-class-information
          :name name
          :abstract-p abstract-p
          :superclasses superclasses
          :slot-specifiers slot-specifiers)
        *planner-class-information*))
```

The function `save-planner-class-information` associates a structure with each type of entity. The user input function looks at the saved information in the structure and then constructs an appropriate prompt for each subclass of entity.

There are two types of planner classes—abstract planner classes and concrete planner classes. Abstract planner classes are basic classes that are never instantiated. Though we store information about these classes, the user will never interact with them. Concrete planner classes are classes for which the user can create instances. We define such classes

by using the macros `def-abstract-planner-class` and `def-concrete-planner-class`, respectively. For example, we redefine locatable entities in the following way:

```
;;; A locatable-entity is one that has an address
;;; and phone number (such as a person or organization).
(def-abstract-planner-class locatable-entity (entity)
  ((address :type address)
   (phone-number :type phone-number)))
```

The information contained in the class definition is the same as before; only the defining form has changed.

How might we best use the stored information from our class definitions in the user interface? We need a new user input function that can obtain stored information, present it to the user in a clear form, and then update the stored information or present new information based on the user's response to the input options. We call this new function `accept`, meaning "accept information."

To determine how this function behaves, let's step through an input session. In this hypothetical interaction, the top-level prompt character (>) indicates what the user sees, the arrow (-->) indicates the *Personal Planner*'s response, and the double-stemmed arrow (==>>) means that the information to the left of the arrow is stored in the variable on the right. Program code is shown in uppercase characters:

```
;; Retrieve and display the menu options.
(ACCEPT '(MENU-MEMBER ADD FIND CHANGE HELP) "command" "commands")

> Select one of the following commands:
  1: ADD
  2: FIND
  3: CHANGE
  4: HELP

  Enter a number or name: 1

--> ADD
```

```
;; Add a new object.  Display the types that can be added.
(ACCEPT `(MENU-MEMBER ,@OBJECT-TYPES) "object type" "object types")

> Select one of the following object types:
  1: PERSON
  2: ORGANIZATION
  3: EVENT
  4: MEETING

  Enter a number or name: ORGANIZATION

-> ORGANIZATION

;; Add an organization.  Retrieve the information required to
;; define an organization by calling a specialized acceptor
;; function for ENTITY objects.
(ACCEPT 'ORGANIZATION)

-> (GET-SLOT-INFORMATION 'ORGANIZATION)
     ((NAME :INITARG :NAME :TYPE STRING)
      (ADDRESS :TYPE ADDRESS)
      (PHONE-NUMBER :TYPE PHONE-NUMBER))     ==>>   slot-info

;; Allow the user to add the organization information
;; slot by slot. Initialize an instance of the organization.
(ACCEPT-INITARGS slot-info "organization")

> Name for this organization: Lucid, Inc.

-> (:NAME "Lucid, Inc.")                     ==>>   initargs

;; Store the instance by name.
(APPLY #'INTERN-INSTANCE initargs)

-> #<Organization "Lucid, Inc.">

;; Prompt for the other slot information by calling
;; specialized acceptors for each slot.
(ACCEPT-NON-INITARG-SLOT-VALUES slot-info "organization")

;; Present the options for an address.
(ACCEPT 'ADDRESS "address for this organization")
```

```
> Address for this organization:
   Street for this address: 707 Laurel Street
   City for this address: Menlo Park
   State for this city: CA
   Zip code for this address: 94025
  Phone number for this organization:
   Area code for this phone number: 415
   Exchange for this phone number: 555
   Local number for this phone number: 8400
   Does this phone number have an extension? (Y or N) N

;; Present the options for a phone number.
(ACCEPT '(MENU-MEMBER VOICE FAX MODEM) "type for this phone number"
                                       "types for this phone number")

> Select one of the following types for this phone number:
  1: VOICE
  2: FAX
  3: MODEM

  Number or name: 1

-> VOICE
```

This user interface is much improved over the one presented in Chapter 2, and it takes advantage of our new strategy.[2] Thus, by using new defining forms, the rest of our revisions are localized to the user interface.

In making these changes, we always first consider the impact they have on our interfaces. We make certain that the operations described by our initial programmatic interface

2. In coding this revised interface, we added another layer of abstraction for classes that are created with `accept`. We created a new superclass called `basic-planner-object`, from which `planner-object` can inherit. Thus, in the final version of the code, `planner-object` is redefined in the following way, which takes advantage of multiple inheritance:

```
;;; BASIC-PLANNER-OBJECT is the parent of all classes defined in
;;; Personal Planner.
(defclass basic-planner-object ()
  ())
;;; Events and entities both inherit from the PLANNER-OBJECT, which
;;; has the PROPERTY-LIST-MIXIN.  We may later add other general
;;; functionality to PLANNER-OBJECT.
(defclass planner-object (basic-planner-object property-list-mixin)
  ())
```

are covered by our revised version, either directly or by being subsumed into other, more abstract operations (see Chapter 2 for our early notes on this interface). Instead of scrapping our initial plan, we simply let it evolve. As the program moves from concept to reality, we discover where we earlier had misconceptions or had not considered all our needs, and then we adjust the program accordingly. Although we modify our data structures and their supporting constructs, our algorithm and the overall behavior of the program remain unchanged.

Chapter 5

COMMENTING AND ORGANIZING THE FILES

5.1 INTRODUCTION

In the process of organizing the material for this book, we found it difficult to decide on the proper place for this discussion of comments and of the physical organization of program files. On the one hand, we wanted these topics to have a prominent place—a chapter of their own. Yet, in making that choice, we took the risk that someone might interpret this segregation as a subtle suggestion that these programming steps happen only after the code is written. To dispel that notion, let us just say explicitly that commenting is an ongoing process that accompanies code development. Similarly, as you write the independent pieces of a program, the way you organize them into files can reflect the modularity of the program design.

Comments are a very important ingredient of a well-written program because they can advertise the intentions underlying the actual state of affairs. This type of information is not only important to others who use and maintain a program but also quite useful to you as the developer. Comments help to remind you of what decisions you have made as well as what you plan to do next. They help you to collect and organize your thoughts. If you must interrupt work on the program, good comments can speed your reorientation to the task.

A well-organized and well-named set of program files forms another type of commentary that makes it easier for you and others to locate pieces of code for updates or revisions. Similarly, organizing a program into its constituent parts makes it easier for you to replace a module without disrupting the organization of the rest of the program.

The discussion of these two components of programming style is somewhat intertwined in the following sections.

5.2 SELF-COMMENTING CODE

The easiest and most natural type of commentary is *self-commenting code*, that is, code that uses meaningful names for functions and variables, includes documentation strings if appropriate, and is logically organized and formatted.

Common Lisp does not impose limits on the size of object names, so you are free to attach as descriptive a name as is reasonable to any symbol. What constitutes a reasonably descriptive name? One that describes in a straightforward manner the main action performed by the construct or the concept embodied by it but that can also be typed quickly and easily. Keep names short enough to allow expressions containing them to be indented consistently with the rest of the program, yet long enough to convey correct information without ambiguity.

Whenever you use a compound name for a Lisp symbol, hyphenate the terms:

```
thisnameisveryhardtoread

ThisNameIsEasierToReadButHarderToType

this-name-is-just-right
```

Hyphens are easy to type, since they don't require the Shift key, and they provide clear and easily readable separations of the name's parts; they also follow established Common Lisp naming conventions. If the symbol names an operation on a particular Lisp type, specify the type as the first part of the name. Common Lisp constructs like `bit-and`, `char-downcase`, `hash-table-count`, and `string-capitalize` follow this convention.

Economy is a useful feature of symbol names. Don't belabor the obvious. For example, in the first version of the *Personal Planner*, we had a function named `rename-entity` that renamed an entity, removed references to the old name in the entity table, and then restored the entity to the table under a new name. A persnickety programmer might have come up with `rename-entity-and-remove-references-to-old-name`; however, most readers of the program would have learned enough from the name `rename-entity`.

Choose names that match the use of a symbol. For example, if you have a temporary variable that represents a disposable, intermediary value, you can deemphasize the variable by using a short name, such as `temp-x` or `temp-y` rather than the more hifalutin' `intermediate-value-x`. The intention of this naming scheme is to imply that the name of this variable is inconsequential—it could be anything. In this way, the meaningful names of the relevant variables in the expression stand out.

Include a documentation string as part of the definition of every user-visible function, macro, and variable. All Common Lisp defining forms, such as `defun`, `defmacro`,

`defvar`, `defparameter`, `defstruct`, and `define-setf-method`, allow documentation strings in their syntax. A documentation string serves as both internal and external commentary—internally, it makes the function or variable self-documenting, and externally, it provides documentation that the user can access with the Common Lisp function `documentation`:

```
> (defvar *planner-binary-pathname-type*
  #+Lucid (first *load-binary-pathname-types*)
  #-Lucid (error "Unknown binary type")
  "The standard pathname type for compiled Lisp files on this
   machine.")
*PLANNER-BINARY-PATHNAME-TYPE*
> (documentation '*planner-binary-pathname-type* 'variable)
"The standard pathname type for compiled Lisp files on this machine."
```

Many of the same Common Lisp defining forms that allow documentation strings also allow you to add declarations as part of the definition. Certain Common Lisp control constructs, such as `do`, `let`, `flet`, `prog`, and `with-open-file`, also support internal declarations. Though the purpose of declarations is to increase the efficiency of compiled code, they have the useful side effect of documenting the compiler assumptions implicit in the definition of the Lisp object containing them.

Use indentation to impose syntax. If you have access to an editor that includes a Lisp-indentation facility, use it to pretty-print your Lisp code. Often Lisp-indenting your code will reveal misplaced parentheses and other syntactic errors. It always makes your code more readable.

A couple of naming conventions are entrenched in Common Lisp. First, the name of a global variable should begin and end with an asterisk to draw attention to its special nature. Second, Lisp predicates should end with a `p`, as described by Steele:

- If the name of the predicate is formed by adding a `p` to an existing name, place a hyphen before the final `p` if the existing name is hyphenated, such as in `hash-table-p`.
- If the existing name has no hyphen, simply append a `p` without a hyphen, such as in `alphanumericp`.
- If the name of the predicate is formed by adding a qualifier to the front of an existing predicate name, join the two names with a hyphen without changing the existing predicate ending, such as in `char-lessp`.

5.3 USEFUL COMMENTARY

A well-written program always contains comprehensive commentary. Comments complement the code; that is, they provide information not readily obtained by quick inspection. Be generous with comments of this nature. Avoid commenting the obvious; this type of comment is not only unnecessary and undesirable, but it also can detract from the useful material. In skimming over the prattle, the reader might miss a well-chosen comment.

The semicolon character (;) introduces most comments in Common Lisp. You need two types of comments in program files—overview comments to introduce the file, and body commentary to explain the body of the code within the file.

5.3.1 Overview Comments

Begin every file with an overview. The overview includes the programmer's name, especially if the file is part of a multiprogrammer project. It also has a description of the contents of the file and how they relate to other parts of the program. The overview can point out dependence that exists between this file and other files and can draw attention to external functions and macros that are used by constructs in the file, as well as any global variables:

```
;;; The following external macros and functions are used
;;; by this file:
;;;     make-my-sequence uses
;;;       extract-element-type (array module)
;;;     my-length uses
;;;       sequence-dimension
;;;       sequence-fill
```

Once the initial draft of the file has stabilized, add an edit history that catalogs any substantive modifications to the file. Finally, include any other information that you think will aid in the proper understanding of the file. By convention, the file overview is marked by a column of triple semicolons on the left margin. Otherwise, the layout of the overview is not important as long as it is easy to read and is used consistently across all the program files.

For example, the file `city.lisp` of the *Personal Planner* begins with the following information:

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: PLAN; Base: 10 -*-
;;;
;;; Define the class CITY for Personal Planner
;;;
;;; Programmer: Eric Benson
;;;
```

```
;;; Edit-History:
;;;
;;; Created: Sun Jan 22 08:56:47 1989
;;; 30-Jan-89 eb: Switched from pcl::initialize to shared-initialize.
;;;
;;; End-of-Edit-History
```

This overview begins with a set of *file options*, which are parsed by GNU Emacs and by Lisp editors in many implementations to determine the editor mode, the syntax of the file buffer, the Lisp package to associate with the buffer, and the numeric base used in the file. It then presents a brief description of the file. The contents of this file form an independent module; they do not depend on the previous evaluation or compilation of any other file. Eric Benson was the programmer, and at the time of this snapshot, he had entered one important change in the file, which is documented in the edit history.

In the top-level file, which contains the main entry point of the program, the overview describes the user-visible behavior of the program in general terms and comments the intent of the program. In addition to the information normally found in a file header, there is a description of the overall structure of the program, so that readers of the program overview can understand how modules relate. If a choice of data structure is relevant to understanding the program, the overview covers the motivations for choosing the data structure and then refers to the file that defines the data structure; the description of the data structure normally accompanies its definition.

Finally, the overview mentions any known bugs in the program and any restrictions that might prevent the program from running correctly (see Section 5.3.2 for a description of the `#|` and `|#` comment characters used in the following example):

```
#|
known bugs:

In the cases where LOAD compiles a source file and loads the
resulting binary, there is no checking of the compile switches.
The compiler may produce an indigestible binary file that LOAD
will blithely try to load.
|#
```

When any of the bugs are fixed or restrictions are lifted, edit the overview to correctly reflect the state of the program.

If the program overview becomes very long, you might want to put it into a separate file with a descriptive file name and then refer to that file in the program's top-level file.

5.3.2 Body Commentary

In addition to the file overview, add comments to the definitions contained in the file. There are four types of body comments, each of which is distinguished by the number of semicolons preceding it.

FOUR SEMICOLONS (;;;;)

Four semicolons denote a subheading in the file; the subheading begins at the left margin. A subheading groups related definitions. For example, the following comment might appear in a program's top-level file:

```
;;;; User-visible definitions
```

The subheading can be a briefly stated topic rather than a complete sentence. Think of it as a headline for the section.

THREE SEMICOLONS (;;;)

Three semicolons denote a description of the succeeding function, macro, or variable definition. The description explains what the particular construct does and any details of the code that are not obvious, especially any hidden side effects. If the arguments to the function or macro are not self-explanatory, the comment might also describe them. However, if the arguments are appropriately named and are used by a local group of functions that call each other without interacting with functions in other files, an argument description is generally not necessary.

If the definition introduces a key data structure of the program, the comment describes how the data structure is organized and how other pieces of the program interact with it. It also gives details of any size or performance restrictions on the data structure.

For comments of this kind, use complete, grammatical sentences, and begin each line at the left margin. These comments do not need to be verbose, only as complete as necessary to explain the operation.

If you find that it is hard to describe what the construct does, ask yourself whether the operation is just plain difficult to perform or whether the construct does not perform the operation in the most straightforward way possible. If the code has no excuse for being obscure, rewrite the code before commenting it.

TWO SEMICOLONS (;;)

Two semicolons denote a description of the succeeding expression. Such comments are meant to briefly clarify implementation details of the expression. Use complete sentences

for these comments, and indent them to the same indentation level as the expression they describe.

ONE SEMICOLON (;)

One semicolon denotes an in-line comment that explains a particular element of the expression on that line. It is especially important to identify any variables on the line whose values might not be clear from the context in which they are used.

Brevity is important for in-line comments, so sentence fragments are acceptable. Try to leave enough space between the code and the comment so that the comment can be spotted easily.

The following example shows how the *Personal Planner* uses some of these types of comments:

```
;;; A STATE is an entity representing a state in the United States,
;;; a Canadian province, or a foreign country.
(defclass state (entity)
  ;; ABBREVIATION is the standard Postal Service two-letter
  ;; abbreviation.
  ((abbreviation :type string :accessor state-abbreviation)
   ;; TIME-ZONE is the predominant time zone in the state.
   ;; When a city is created, this time zone is used unless one
   ;; is supplied.
   (time-zone :type integer)))
```

Note that we type the names of the constructs being described in capital letters for emphasis. See Appendix A for a complete example of our commenting style.

Common Lisp also provides the characters #| and |# for enclosing a block of comments or for commenting out code or data. Since these comment characters are not as easy to notice as leading semicolons, this type of comment generally indicates that the commented condition is temporary. For example, in Section 5.3.1, we used this type of comment when describing a bug that we intended to fix in a later version of the code. Another common technique is to comment out a block of code so that it can be skipped over in a file, as during debugging.

5.4 CODE FILE ORGANIZATION

It is a good idea to break your program up into functional parts by splitting modules pertaining to certain data structures and tasks into separate, independent files, with one module per file. Segregate any implementation-specific information from the portable

pieces of the program. Use file names that describe the modules contained in the file and that indicate whether the information in the file depends on a particular implementation. This technique makes the program easier to manage because the pieces are easier to locate and the nonportable code is identified. It also encourages task sharing among programs because the modules are convenient to drop into different contexts.

Consider a directory listing for the *Personal Planner*:

```
accept.lisp                    file-save.lisp
address-object.lisp            interned-mixin.lisp
city-object.lisp               load.lisp
database.lisp                  meeting-object.lisp
date-constants.lisp            organization-object.lisp
date-parse.lisp                package.lisp
date-write.lisp                person-object.lisp
display.lisp                   phone-number-object.lisp
edit.lisp                      planner-object.lisp
entity-object.lisp             plist.lisp
event-object.lisp              state-object.lisp
extra-types.lisp               time-zone.lisp
file-constants.lisp            toplevel.lisp
file-restore.lisp              type-dispatch.lisp
```

The class definitions for each object are maintained in separate files, whose file names specify the class. Thus, we can see at an instant that the program manipulates addresses, cities, entities, events, interned mixins, meetings, organizations, persons, the planner object, phone numbers, and states. These class files include any methods associated with the class, so by examining a class file, we can determine both the structure and the behavior of the class. If we want to add a new behavior to the class, we simply add a new method to the file. Whenever we add a new class to the system, we introduce a new object file.

The names `accept`, `display`, and `edit` denote ways to accept, display, and change user input. There are files for manipulating dates and types as well as a database file and files for saving and restoring that database. And there are a few miscellaneous utilities, such as the files defining time zones and states. None of these file names indicates that the code depends on a specific type of hardware or a software implementation.

5.4.1 Administrative Files

Most programs also require some administrative tasks to set up the run-time environment of the program. For example, there should be some way of ensuring that the files for the program are properly initialized and that they are all loaded into the system before the program is run. There should be some package specifications to classify the symbols of

the program. There might be some declaration files to assist in the efficient compilation of the program. Finally, if you think that someone reading your directory might be confused by its contents, you might include a `README` documentation file that explains the files pertaining to the program and how to use them.

Our directory shows a couple of administrative files, `load.lisp` and `package.lisp`. Note that these files begin with an `(in-package "USER")` statement, while all the functional files for the program are organized into the `planner` package.

5.4.1.1 COMPILATION AND LOAD FILES

The file `load.lisp` is the top-level file that sets up the *Personal Planner*. It compiles and loads all the program files when we enter the command `(load-planner)` to the Lisp top-level prompt:

```
(in-package "USER")

;;; A simple mechanism for compiling and loading these files in the
;;; correct order.  The variables *PLANNER-SOURCE-PATHNAME-TYPE*
;;; and *PLANNER-BINARY-PATHNAME-TYPE*, and the macro
;;; PLANNER-BATCH-COMPILATION-WRAPPER will have to be modified when
;;; this program is ported to a new Lisp implementation.

;;; There is relatively little compile/load order dependence in
;;; these files.  Superclasses are always defined before their
;;; subclasses, despite the fact that this is not required by CLOS.

(defvar *planner-files*
  '("package"          ;Setting up the PLANNER package required
                       ;by all files
    "plist"
    "extra-types"      ;Miscellaneous extra types used
                       ;in later files
    "planner-object"   ;DEF-PLANNER-CLASS needed to compile
                       ;many files
    "interned-mixin"
    "entity-object"
    "time-zone"        ;Needed before calling MAKE-STATE
    "state-object"     ;Needed before loading states
    "address-object"
    "phone-number-object"
    "city-object"
    "person-object"
```

```
    "organization-object"
    "event-object"
    "meeting-object"
    "date-constants"  ;Constants needed by DATE-WRITE
                      ;and DATE-PARSE
    "date-write"
    "date-parse"
    "toplevel"        ;Command interpreter
    "type-dispatch"
      "accept"          ;Generalized input based on data type
      "edit"            ;Generalized object editing
      "display"         ;Generalized display
      "database"
      "file-constants"  ;Constants used for file saving and restoring
      "file-save"       ;Save any object to a file
      "file-restore"    ;Read any object from a file)
    "A list of the files in Personal Planner.")

(defvar *planner-source-pathname-type*
  #+Lucid (first *load-source-pathname-types*)
  #-Lucid (error "Unknown source type")
  "The standard pathname type for Lisp source files
   on this machine.")

(defvar *planner-binary-pathname-type*
  #+Lucid (first *load-binary-pathname-types*)
  #-Lucid (error "Unknown binary type")
  "The standard pathname type for compiled Lisp files
   on this machine.")

(defmacro planner-batch-compilation-wrapper (&body body)
  "This macro gets wrapped around a sequence of calls to
   COMPILE-FILE so that per-file warnings, such as those
   about unknown functions, will be delayed until the end."
  (let ((system-dependent-wrapper
         #+Lucid 'with-deferred-warnings
         #-Lucid 'progn))
    `(,system-dependent-wrapper ,@body)))

(defun load-planner
    (&optional (action :binary)
               (directory *default-pathname-defaults*))
  "Load Personal Planner.  ACTION can be any of :BINARY,
   :SOURCE, or :COMPILE. If ACTION is :BINARY, load compiled
```

```
   files; if :SOURCE, load source files; if :COMPILE, compile
   the source files and load the resulting binaries.
   DIRECTORY is the location of the files."
  (let ((source-default
         (make-pathname :defaults directory
                        :type *planner-source-pathname-type*))
        (binary-default
         (make-pathname :defaults directory
                        :type *planner-binary-pathname-type*)))
;; Just load, or compile and then load, the files in the order
;; given.
(planner-batch-compilation-wrapper
  (dolist (file *planner-files*)
    (let ((source-file (merge-pathnames file source-default))
          (binary-file (merge-pathnames file binary-default)))
      (case action
        (:compile
         (compile-file source-file)
         (load binary-file))
        (:binary
         (load binary-file))
        (:source
         (load source-file)))))))
```

The *Personal Planner* is small and simple enough so that we can specify load-time or compile-time dependence by simply listing the files in the proper order in the global variable `*planner-files*`:

- The file `package.lisp` sets up the correct namespace for the rest of the program files.
- The file `extra-types.lisp` contains `deftype` expressions used by constructs in the files `entity.lisp`, `event.lisp`, and `accept.lisp`; thus, it must be compiled early.
- The file `planner-object.lisp` contains the defining macros for abstract and concrete plan classes, which must be compiled before any entity files can compile correctly.
- The file `time-zone.lisp` contains a definition for a time zone type, which is needed for `state-object.lisp` to compile correctly.
- The file `date-constants.lisp` contains definitions that must be compiled before constructs in `date-write.lisp` and `date-parse.lisp` can be compiled.
- The file `file-constants.lisp` contains definitions that must be compiled before constructs in `file-save.lisp` and `file-store.lisp` can be compiled.

Thereafter, the ordering shown by `*planner-files*` is for our own convenience, since Lisp does not impose any requirements upon the loading or compilation order of subclasses, superclasses, methods, and generic functions. [1] As a form of self-documentation, we group the classes and subclasses together.

The function `load-planner` lets us specify whether to load source files or binary files, or to compile the source files and then load the resulting binary files. The global variables `*planner-source-pathname-type*` and `*planner-binary-pathname-type*` distinguish source and binary files in an implementation-dependent way. If the *Personal Planner* is loaded into a implementation that does not support the underlying variables `*load-source-pathname-types*` and `*load-binary-pathname-types*`, an error is signaled. Upon examining the file to correct the error, the maintainer of the program can determine the source of the problem by reading the porting comment at the top of the file.

The file `toplevel.lisp` contains the user's top level. Once `load.lisp` has been loaded into a Lisp image and the function `load-planner` has completed execution, the user can start the *Personal Planner* by entering the command `personal-planner` to the Lisp prompt.

This type of solution for loading and compiling our program cannot work for large programs or for programs that have complex load-time and compile-time dependence. For these, you would need to use a more sophisticated system-building tool that allows you to specify file or module dependence more accurately. Many Lisp systems include such a facility, but there is no standardized version.

5.4.1.2 PACKAGE FILES

The Common Lisp package system is a means of organizing the names and symbols in a program. When used correctly, it prevents the accidental misuse or redefinition of program symbols by allowing you to associate a given symbol name with a specific package.

It is always a good idea to specify a *home package* for program definitions by beginning each program file with an `in-package` statement. The definitions then reside in that package and are independent of definitions in other packages. Thus, if your program contains the symbol `my-symbol` and there is a symbol of the same name that has a different definition in another package, Lisp will not confuse the two definitions during the execution of your program—it will always use the symbol defined in your program's home package. Any modifications you make to the symbol in your package have no effect on the symbol in the other package. Users of your program need never know that two symbols

1. By contrast, a structure must be defined before you can include it in another structure or compile any usage of its accessors, constructors, or `setf` methods.

of the same name exist because they will only see the symbols specified by the program's package. Thus, using a package emphasizes the program modularity.

Defining symbols in one package does not prevent you from using symbols in other packages. You can access any other symbol by specifying that your package use its package or by using package qualification when you call on the symbol. See Steele for information about using symbols from other packages. Note, however, that there is a better ordering for package operations than the one hinted at by Steele's mnemonic sentence "Put in seven extremely random user interface commands." The following order for specifying Common Lisp package operations is recommended in the *Lucid Common Lisp Advanced User's Guide*:[2]

- `in-package` (or an equivalent use of `make-package`)
- `shadow` (and possibly `shadowing-import`)
- `use-package`
- `import` (and possibly `shadowing-import`)
- `export`

Finally, use double-colon qualification only as a last resort; it is a very dangerous type of access that violates package conventions and that can be avoided in most situations.

When your program consists of more than one or two files, create a package administrative file that specifies the package requirements for the program. Make certain that this file is loaded first, before anyone tries to access the program symbols.

The *Personal Planner* uses the file `package.lisp` to set up its namespace:

```
(in-package "USER")

;;; The PLANNER package uses the LISP, LUCID-COMMON-LISP, and CLOS
;;; packages.  The LISP package contains the symbols documented in
;;; Steele, the LUCID-COMMON-LISP package contains the Lucid
;;; extensions to Common Lisp, and the CLOS package contains the
;;; CLOS constructs we need.  We use :IMPORT-FROM explicitly to avoid
;;; problems with CLOS::CLASS-PROTOTYPE.
(defpackage "PLANNER"
  (:use "LISP" "LCL" "CLOS")
  (:import-from "CLOS" "CLASS-PROTOTYPE"))
```

2. This preferred ordering is reflected in a new package construct, the macro **defpackage**, which was voted into the Common Lisp standard in January 1989. This macro provides a functional way of setting up the naming environment for Lisp programs.

This file makes certain that the `planner` package exists and is compiled before any of the other files are loaded or compiled. This check is necessary because of a subtle property of the function `in-package`.

All our program files begin with the following statement:

```
(in-package "PLANNER")
```

Thus, when that file is loaded, `in-package` changes the current package to the `planner` package.

The `planner` package uses the symbols in the `lisp`, `lcl`, and `clos` packages. Thus, when our program is executing and comes across a symbol, it looks first in the `planner` package for a definition; if the symbol is not in that package, it looks in the `lisp`, `lcl`, and `clos` packages respectively. If the symbol is not defined in any of these packages, an error is signaled.

We have imported an internal symbol of the `clos` package explicitly into the `planner` package to avoid using double-colon qualification. The `:import-from` statement advertises our intent to rely on a symbol that is not otherwise generally available, even to packages that use the `clos` package.

5.4.1.3 DECLARATION FILES

If your program is very large, you might need to establish declaration files to ensure proper compilation and loading of program constructs. Two types of declaration files can be useful—one that contains the building blocks of a program that are only needed at compile-time, and one that contains constructs that need to be available at both run- and compile-time. Use declaration files when your program includes the following constructs but cannot restrict their use to the file that contains their definitions:

- `defvar`, `defparameter`, and `defconstant` definitions
- Macros
- Functions proclaimed in-line
- `defstruct` definitions and constructors and accessors that use them

If you have a large program that uses CLOS, you might want to keep all of your `defclass` definitions in one file to advertise which types of objects your program manipulates. Note, however, that there are no requirements in CLOS about the order in which classes, methods, and generic functions are loaded and compiled.

Declaration files serve as conveniences to the compiler; they help to set up the proper compile-time and load-time environment. However, there is a serious drawback to using a declaration file—it disrupts local transparency in the program. Thus, to completely understand the behavior of certain parts of the program, the maintainer must consult at least two files instead of simply reviewing a self-contained block of code. As much as is possible, organize a program to reflect its structure. However, if you must sacrifice local transparency for the sake of the compiler, it is best to draw attention to the fact by putting the compiler-dependent information in one or more declaration files that can be loaded after the package administration file of your program.

5.4.2 Organizing Individual Files

In general, we suggest you follow these guidelines for organizing an individual program file:

- Begin each file with a descriptive header that contains an edit history of the file.
- Follow the header with an `in-package` statement that identifies the package for the file.
- If you are not using a declaration file, put global variables and parameters after the package statement so that other programmers can easily locate their definitions.
- Comment the definitions carefully. Identify all top-level functions.
- Where possible, follow the principle of local transparency within the file. Try to organize your code so that the local flow of control and the binding of lexically referenced variables are clear, regardless of the form the data take.
- Allow the contents of the file to reflect the modularity of the program. Do not intermingle modules in a file.

Chapter 6

DEBUGGING

6.1 THE STYLE OF DEBUGGING

Though it might seem that debugging is not a style issue, it should be clear that a well-written program is not always a correct program. Certainly some debugging techniques are more effective than others, which in a way makes them stylistically more pleasing. For these reasons, debugging is a worthwhile subtopic for any discussion of programming style.

It is not as easy to come up with debugging style guidelines as it might be to come up with pointers for writing a clean, efficient, and understandable program. The success or failure of a debugging session often hinges upon where the debugging process leads as a result of judgment choices that you make interactively in response to a Lisp debugger. Since the debugger's alternatives are tied to the dynamic environment, there is no way to guess at what choices it might present. Nor is there any possible formula for what constitutes good judgment without knowing the full context of the situation. Nevertheless, a few general recommendations can be made for many debugging situations, and these general observations are presented in this chapter.

Of course, you have an advantage if you have planned your program well and have followed that plan, since you know what parts of your program depend on others and what results your program should produce. If you have paid attention to programming style while writing the code and have used effective commentary, you will have an easier time tracking down any problems found by the Lisp debugger or by users of your program. Thus, the first steps of good debugging style are the components of good programming style. The following sections describe other tools and techniques that improve the debugging process.

6.2 LISP DEBUGGING TOOLS

In addition to an interactive environment that allows you to build and test programs incrementally, Common Lisp contains several constructs that provide information about the definition and behavior of Lisp functions and objects. It also provides a mechanism that allows you to define, detect, and recover from error situations that might arise in your own program. The complete functionality of these constructs might differ slightly across implementations.

To benefit from the built-in debugging capability of Common Lisp, you must become familiar with these tools—how they are similar, how they are dissimilar, how they respond to different types of input, and when they are most useful. For the purposes of this discussion, we divide the debugging tools into two broad categories:

- Interactive debugging tools
  These tools are meant to be used either at the top level of Lisp or inside any other Lisp facility that has a read-eval-print loop; do not include them as part of a function definition.
- Programmatic debugging tools
  These debugging tools are meant to be built into program definitions. Constructs in the error-handling facility that became a part of Common Lisp in 1988 fall into this category.

The following sections describe the constructs in these categories.

6.2.1 Interactive Debugging Tools

The interactive constructs provide general information about program objects and behavior. Unlike the programmatic debugging tools, these tools cannot be customized for your application. Nevertheless, they provide a good first line of defense against bugs and can be used inside a Lisp debugger to explore programs at depth.

DESCRIBE

The function `describe` takes any Lisp object as an argument and prints out general information about the object. This function is useful for gaining a broad understanding of the object. Though the output is implementation dependent, it is generally type specific and often includes details about the structure of the object. For example, `describe` might show the components of symbols.

Suppose that you have written and compiled the following function in the current Lisp environment:

```
;;; This function modifies its list argument so that all elements
;;; from embedded lists are moved to the list's top level.  The
;;; denesting mechanism is CAR-CDR recursion.
(defun flatten (lis)
  (cond ((null lis) nil)
        ((atom (car lis)) (cons (car lis) (flatten (cdr lis))))
        (t (append (flatten (car lis)) (flatten (cdr lis))))))
```

When you invoke `describe` with the symbol `flatten` as an argument, you learn the following information:

```
> (describe 'flatten)
FLATTEN is a symbol.  Its home package is USER.
Its global function definition is
  #<Compiled-Function FLATTEN 92451F>.
#<Compiled-Function FLATTEN 92451F> is a compiled function.
Its argument list is (LIS).
The code for this function is 486 bytes long.
The function definition for FLATTEN is at top level.
It has the following property list elements:
  For indicator SOURCE-CODE the value
  #<Interpreted-Function (NAMED-LAMBDA FLATTEN (LIS)
                           (BLOCK FLATTEN (COND # # #))) 920987>.
```

The output from `describe` identifies the type and package for `flatten` as well as information about its function binding. Note how the information changes when you supply the functional definition of `flatten` as the argument:

```
> (describe #'flatten)
#<Compiled-Function FLATTEN 92451F> is a compiled function.
Its argument list is (LIS).
The code for this function is 486 bytes long.
```

In this case, only the global function definition of the symbol is described.

INSPECT

The function `inspect` takes the output from `describe` several steps further; it allows you to descend through the description of the object to find out different levels of information. This function is useful for discovering implementation details about the object. For example, consider the output when `inspect` is invoked on the symbol `flatten`, which represents an uncompiled version of the function:

```
> (inspect 'flatten)     ;This version of FLATTEN has not been
                         ;compiled.
#<Symbol 55FF95>

[0: NAME] "FLATTEN"
[1: VALUE] Unbound
[2: FUNCTION] #<Interpreted-Function (NAMED-LAMBDA FLATTEN (LIS)
                  (BLOCK FLATTEN (COND # # #))) 920987>
[3: PLIST] (LUCID::ADVICE-ORIGINAL-DEF NIL LUCID::ADVICE-DATA NIL
            SOURCE-CODE #<Interpreted-Function
                           (NAMED-LAMBDA FLATTEN (LIS)
                             (BLOCK FLATTEN (COND # # #))) 920987>)
[4: PACKAGE] #<Package "USER" 515CCB>
```

While `describe` presents its information in a form that is clear to the naive user, `inspect` presents a lower-level description that might be useful to a more experienced Lisp programmer. To find out more about the functional binding of this symbol, you can choose to inspect it interactively by supplying predefined directives to `inspect`. In the following example, the `inspect` prompt is >>; the directives are the numbers that `inspect` uses to identify the different components of its output:

```
>> 2
#<Interpreted-Function (NAMED-LAMBDA FLATTEN (LIS)
  (BLOCK FLATTEN (COND # # #))) 920987>

[0: CODE] #<Code 12E82B>
[1: NAME] (NAMED-LAMBDA FLATTEN (LIS) (BLOCK FLATTEN (COND # # #)))
[2: FLAGS] 76
[3: ARGLIST] (LIS)
[4] #<Structure LUCID::LCODE-CLOSURE 920973>
[5] LUCID::LRUN-INSIDE-LAMBDA
```

To get a closer look at the code for the function definition, you can descend a few more levels by entering the appropriate directives to `inspect`:

```
>> 1
#<List 9208C1>

[0] NAMED-LAMBDA
[1] FLATTEN
[2] (LIS)
[3] (BLOCK FLATTEN (COND (# NIL) (# #) (T #)))
```

```
>> 3
#<List 9208A1>

[0] BLOCK
[1] FLATTEN
[2] (COND ((NULL LIS) NIL) ((ATOM #) (CONS # #)) (T (APPEND # #)))
>> 2
#<List 920721>

[0] COND
[1] ((NULL LIS) NIL)
[2] ((ATOM (CAR LIS)) (CONS (CAR LIS) (FLATTEN #)))
[3] (T (APPEND (FLATTEN #) (FLATTEN #)))
>> 3
#<List 9207D1>

[0] T
[1] (APPEND (FLATTEN (CAR LIS)) (FLATTEN (CDR LIS)))
>> :q
(T (APPEND (FLATTEN (CAR LIS)) (FLATTEN (CDR LIS))))
```

In the following example, note the difference in the information returned by the first and second interactive levels of `inspect` after `flatten` is compiled with a `safety` level of 3. The information for the compiled version is more condensed than that of the interpreted version:

```
> (compile 'flatten)
FLATTEN
> (inspect 'flatten)
#<Symbol 55FF95>

[0: NAME] "FLATTEN"
[1: VALUE] Unbound
[2: FUNCTION] #<Compiled-Function FLATTEN 92B5BF>
[3: PLIST] (LUCID::ADVICE-ORIGINAL-DEF NIL LUCID::ADVICE-DATA NIL
            SOURCE-CODE
              #<Interpreted-Function (NAMED-LAMBDA FLATTEN (LIS)
                (BLOCK FLATTEN (COND # # #))) 920987>)
[4: PACKAGE] #<Package "USER" 515CCB>
>> 2
#<Compiled-Function FLATTEN 92B5BF>

[0: CODE] #<Code 41463B>
[1: NAME] FLATTEN
[2: FLAGS] 4
```

```
[3: ARGLIST] (LIS)
[4] FLATTEN
[5] APPEND
>> :q
#<Compiled-Function FLATTEN 92B5BF>
```

The directives that `inspect` understands are specific to each implementation but might include commands for moving around the stack, for changing the value of an object's slot on the stack, for reformatting the messages shown by `inspect`, and for returning to the top level of Lisp. Refer to the documentation for your implementation for the options that are defined for `inspect`.

APROPOS, APROPOS-LIST

The functions `apropos` and `apropos-list` search through the Lisp environment to see if the print names of any symbols contain the string or match the symbols specified as their first argument. These functions are useful for preventing name conflicts because they can determine whether a name is currently in use. They differ in the amount of information they return; `apropos` generally provides a description of the symbols it finds, whereas `apropos-list` merely lists the symbols.

```
> (apropos 'flatten)
FLATTEN, Function (LIS)
> (apropos-list 'flatten)
(FLATTEN)
```

Unless you specify a single package, the search covers all packages.

```
> (apropos 'trace "LISP")
*TRACE-OUTPUT*, Bound
UNTRACE, Macro
TRACE, Macro
> (apropos 'trace)
:TRACE, Keyword
*TRACE-COLUMNS-PER-LEVEL*, Bound
*TRACE-BAR-P*, Bound
*TRACE-VALUES*, Bound
*TRACE-ARGLIST*, Bound
*TRACE-LEVEL*, Bound
*TRACE-NEW-DEFINITIONS*, Bound
*MAX-TRACE-INDENTATION*, Bound
*TRACED-FUNCTION-LIST*, Bound
*TRACE-OUTPUT*, Bound
UNTRACE, Macro
TRACE, Macro
```

```
> (apropos-list 'trace)
(TRACE UNTRACE *TRACE-OUTPUT* *TRACED-FUNCTION-LIST*
*MAX-TRACE-INDENTATION* *TRACE-NEW-DEFINITIONS* *TRACE-LEVEL*
*TRACE-ARGLIST* *TRACE-VALUES* *TRACE-BAR-P*
*TRACE-COLUMNS-PER-LEVEL* :TRACE)
```

TRACE

The macro `trace` lets you examine the dynamic behavior of a function, that is, follow a function in action. Unlike `describe`, `inspect`, `apropos`, and `apropos-list`, `trace` does not work with any Lisp object—you can only trace functions. Whenever you specify a function name as an argument to `trace`, information about that function is printed to a stream whenever the function is invoked. The variable `*trace-output*` contains the stream that receives the tracing information; by default, the stream that is the value of `*terminal-io*` receives tracing output. The information returned is determined by the particular implementation but usually includes the values of the function's parameters upon entry to the function and the value returned when the function exits:

```
> (trace flatten)
(FLATTEN)
> (flatten '((a (b (c)))))
1 Enter FLATTEN ((A (B #)))
| 2 Enter FLATTEN (A (B (C)))
|   3 Enter FLATTEN ((B (C)))
|    | 4 Enter FLATTEN (B (C))
|    |   5 Enter FLATTEN ((C))
|    |    | 6 Enter FLATTEN (C)
|    |    |   7 Enter FLATTEN NIL
|    |    |   7 Exit FLATTEN NIL
|    |    | 6 Exit FLATTEN (C)
|    |    | 6 Enter FLATTEN NIL
|    |    | 6 Exit FLATTEN NIL
|    |   5 Exit FLATTEN (C)
|    | 4 Exit FLATTEN (B C)
|    | 4 Enter FLATTEN NIL
|    | 4 Exit FLATTEN NIL
|   3 Exit FLATTEN (B C)
| 2 Exit FLATTEN (A B C)
| 2 Enter FLATTEN NIL
| 2 Exit FLATTEN NIL
1 Exit FLATTEN (A B C)
(A B C)
> (untrace)
(FLATTEN)
```

As this example shows, using `trace` is a good way to find out how recursion works. However, tracing is also useful with nonrecursive functions, especially for showing how functions interact. Note that a trace does not provide information about functions called by the traced function unless those functions are also being traced:

```
;;; This function takes an integer argument and returns the closest
;;; prime number that is greater than or equal to the argument.
> (defun next-prime (num)
    (do ((result num (1+ result)))
        ((primep result) result)))
NEXT-PRIME

;;; This predicate determines, with the help of the function
;;; CHECK-DIVISIONS, whether a number is prime.
> (defun primep (n)
    (if (check-divisions n (floor (sqrt n))) t nil))
PRIMEP

;;; This function returns NIL if a number is evenly divisible by any
;;; number other than 1.
> (defun check-divisions (x count)
    (cond ((<= count 1) t)
          ((zerop (mod x count)) nil)
          (t (check-divisions x (1- count)))))
CHECK-DIVISIONS
> (trace next-prime)
(NEXT-PRIME)
> (next-prime 24)
1 Enter NEXT-PRIME 24
1 Exit NEXT-PRIME 29
29
```

To see more information, add `primep` to the traced function list:

```
> (trace primep)
(PRIMEP)
> (next-prime 27)
1 Enter NEXT-PRIME 27
| 2 Enter PRIMEP 27
| 2 Exit PRIMEP NIL
| 2 Enter PRIMEP 28
| 2 Exit PRIMEP NIL
| 2 Enter PRIMEP 29
| 2 Exit PRIMEP T
1 Exit NEXT-PRIME 29
29
```

For the complete interaction, trace `check-divisions` as well:

```
> (trace check-divisions)
(CHECK-DIVISIONS)
> (next-prime 28)
1 Enter NEXT-PRIME 28
| 2 Enter PRIMEP 28
|   3 Enter CHECK-DIVISIONS 28 5
|   | 4 Enter CHECK-DIVISIONS 28 4
|   | 4 Exit CHECK-DIVISIONS NIL
|   3 Exit CHECK-DIVISIONS NIL
| 2 Exit PRIMEP NIL
| 2 Enter PRIMEP 29
|   3 Enter CHECK-DIVISIONS 29 5
|   | 4 Enter CHECK-DIVISIONS 29 4
|   |   5 Enter CHECK-DIVISIONS 29 3
|   |   | 6 Enter CHECK-DIVISIONS 29 2
|   |   |   7 Enter CHECK-DIVISIONS 29 1
|   |   |   7 Exit CHECK-DIVISIONS T
|   |   | 6 Exit CHECK-DIVISIONS T
|   |   5 Exit CHECK-DIVISIONS T
|   | 4 Exit CHECK-DIVISIONS T
|   3 Exit CHECK-DIVISIONS T
| 2 Exit PRIMEP T
1 Exit NEXT-PRIME 29
29
> (untrace)
(NEXT-PRIME PRIMEP CHECK-DIVISIONS)
```

Tracing continues for a function until you turn it off with the macro `untrace`.

STEP

The macro `step` provides tracing of a finer granularity. It allows you to single-step through the execution of a function. The following example shows the results of stepping through `next-prime`:

```
> (next-prime 28)
  (BLOCK NEXT-PRIME (DO (# #) (# RESULT))) -> ?
:n              Evaluate current expression in step mode.
:s              Evaluate current expression without stepping.
:u              Evaluate current expression without stepping
                   and go up one level of stepping.
```

```
:x              Finish evaluation, but turn Stepper off.
:p              Print current expression.
:pp             Pretty-print current expression.
:b              Enter the Debugger.
:q              Exit to Top Level.
:h or ?         Print this text.
```

Like `inspect`, most implementations of `step` provide a listing of directives when you enter a question mark at the step prompts, which are given as -> and --> in this example; the prompt --> indicates that the displayed form is a macro expansion:

```
(BLOCK NEXT-PRIME (DO (#) (# RESULT))) -> :n
  (DO ((RESULT NUM #)) ((PRIMEP RESULT) RESULT)) --> :pp

(DO ((RESULT NUM (1+ RESULT)))
    ((PRIMEP RESULT)
     RESULT)) --> :n
      (LET ((RESULT NUM)) (TAGBODY (GO #:G4) #:G3 (PSETQ RESULT #)
#:G4 (IF # #) (RETURN RESULT))) --> :n
        NUM = 28
        (TAGBODY (GO #:G4) #:G3 (PSETQ RESULT (1+ RESULT)) #:G4
(IF (NOT #) (GO #:G3)) (RETURN RESULT)) -> :n
          (GO #:G4) -> :n
          (PROGN (IF (NOT #) (GO #:G3)) (RETURN RESULT)) -> :n
            (IF (NOT (PRIMEP RESULT)) (GO #:G3)) -> :n
              (NOT (PRIMEP RESULT)) -> :n
                (FUNCTION NOT) -> :n
                #<Compiled-Function NOT 4344F7>
                (PRIMEP RESULT) -> :n
                  (FUNCTION PRIMEP) -> :n
                  #<Interpreted-Function (NAMED-LAMBDA PRIMEP (N)
(BLOCK PRIMEP (IF # T NIL))) 9347AF>
                  RESULT = 28
                  (BLOCK PRIMEP
                    (IF (CHECK-DIVISIONS N #) T NIL)) -> :pp
(BLOCK PRIMEP
  (IF (CHECK-DIVISIONS N (FLOOR (SQRT N)))
      T
      NIL)) -> :n
```

Here `next-prime` enters `primep`. Unlike `trace`, `step` follows all interpreted expressions dynamically within the expression being stepped; however, it does not single-step through invocations of compiled functions. As stepping continues, the output shows the execution of `primep` and the entrance and execution of `check-divisions`:

```
(IF (CHECK-DIVISIONS N (FLOOR #)) T NIL) -> :n
  (CHECK-DIVISIONS N (FLOOR (SQRT N))) -> :n
    (FUNCTION CHECK-DIVISIONS) -> :n
    #<Interpreted-Function (NAMED-LAMBDA CHECK-DIVISIONS (X COUNT)
(BLOCK CHECK-DIVISIONS (COND # # #))) 93391F>
    N = 28
    (FLOOR (SQRT N)) -> :n
      (FUNCTION FLOOR) -> :n
      #<Compiled-Function FLOOR 43106F>
      (SQRT N) -> :n
        (FUNCTION SQRT) -> :n
        #<Compiled-Function SQRT 45585F>
        N = 28
      5.291502622129181
    5  0.2915026221291814 3
    (BLOCK CHECK-DIVISIONS (COND (# T) (# NIL) (T #))) -> :pp
(BLOCK CHECK-DIVISIONS
  (COND ((EQUALP COUNT 1) T)
        ((ZEROP (MOD X COUNT)) NIL)
        (T (CHECK-DIVISIONS X (1- COUNT))))) -> :s
                      NIL
                    NIL
                      NIL
                    NIL
                      NIL
                    T
                    (GO #:G3) -> :n
              (PSETQ RESULT (1+ RESULT)) --> :s
              NIL
              (PROGN (IF (NOT #) (GO #:G3)) (RETURN RESULT)) -> :s
    29
  29
29
> (untrace)
(NEXT-PRIME)
```

As this example shows, single-stepping involves looking at each Lisp object in turn and examining all the Lisp objects it comprises, including internal constructs. This level of detail is often tedious, but single-stepping remains an instructive method for learning about the complete run-time behavior of a function. As with `trace`, you turn off single-stepping with `untrace`.

Note that `inspect` and `step` provide their own versions of read-eval-print loops; thus,

you can type any Lisp expression to their prompts and expect meaningful results, as this example shows:

```
> (defun say-hi ()
    (prin1 "hi"))
SAY-HI
> (inspect #'say-hi)
#<Interpreted-Function (NAMED-LAMBDA SAY-HI NIL
 (BLOCK SAY-HI (PRIN1 hi))) 921E3F>

[0: CODE] #<Code 12E82B>
[1: NAME] (NAMED-LAMBDA SAY-HI NIL (BLOCK SAY-HI (PRIN1 "hi")))
[2: FLAGS] 76
[3: ARGLIST] NIL
[4] #<Structure LUCID::LCODE-CLOSURE 921E2B>
[5] LUCID::LRUN-INSIDE-LAMBDA

>> (step say-hi)
#<List 921F69>

[0] SAY-HI

>> (say-hi)
  (BLOCK SAY-HI (PRIN1 "hi")) -> (describe #'say-hi)

;;; Using unadvised definition of SAY-HI
#<Interpreted-Function (NAMED-LAMBDA SAY-HI NIL
 (BLOCK SAY-HI (PRIN1 "hi"))) 921E3F> is an interpreted function.
Its argument list is NIL.
The code for this function is 1684 bytes long. -> :n
    (PRIN1 "hi") -> :s
"hi"
    "hi"
  "hi"
"hi"

>> (untrace)
#<List 9224B1>

[0] SAY-HI
>> :q
(SAY-HI)
```

Of course, the meaningfulness of the output is proportionate to its capacity.

Steele's section "Debugging Tools" in the chapter "Miscellaneous Features" contains the macro `time` and the functions `room`, `ed`, and `dribble` in addition to the constructs already described. Having an editor that understands Lisp syntax and that can be entered dynamically from Lisp is an invaluable tool, one whose impact extends well beyond debugging. The debugging value of the other constructs Steele mentions in this section is limited.

6.2.2 Programmatic Debugging Tools

A hallmark of good programming style in any language is to include error-checking mechanisms within the code. The intent of these error checkers is to allow *graceful degradation* of the program; that is, they warn the user and take some meaningful and appropriate action whenever they encounter an error.

The Common Lisp standard has always included a set of error-checking constructs that allow you not only to signal when an error occurs but also to enter the debugger and sometimes to continue from the error. Since 1988, the standard has included a complete error-handling facility that allows you to define conditions in your program that should be signaled, handlers to take action if the condition is detected, and restarts to proceed past the error condition if possible. This facility is completely integrated with the error-checking constructs that preceded it.

The following sections describe the original set of error constructs and briefly review the Common Lisp error-handling facility.

6.2.2.1 THE TRADITIONAL TOP TEN

In accordance with recommendations made in the standard, most implementations provide an interactive debugger. These debuggers have their own read-eval-print loop that can evaluate any Lisp expression. This capability allows you to redefine erroneous functions on the fly. In addition, it lets you change the values of local variables dynamically. Debuggers generally provide access to the Lisp stack as well, so that you can trace the path to the error. Many implementations now build their Lisp debuggers from the error-handling facility described in Section 6.2.2.2.

Like `inspect` and `step`, most debuggers return information about available debugger commands when you type a question mark to their prompt. In the following example, the debugger prompt is shown as `->`:

```
-> ?
<N>      Invoke debugger option <N>
:A       Abort to most recent Abort Restart.
:A :T    Abort to earliest recent Abort Restart (usually Top Level).
:C       Continue, invoking most recent Continue Restart.
:N       Go down one frame (towards the caller).
:N k     Go down k frames.
:P       Go up one frame.
:P k     Go up k frames.
:>       Go to the bottom of the stack (towards Lisp Top Level).
:<       Go to the top of the stack.
:B       Backtrace to the bottom of the stack.
:D       Display current frame and show options.
:V       Verbosely display current frame, showing up to 32 local variables.
:L k     Print local variable numbered k and set * to its value.
:F       Retry execution of the current frame.
:R       Return a value to the caller of the current frame.
Expressions that are not debugger commands will be evaluated and printed.
For additional help on the full set of debugger commands use :??.
->
```

Combined with an implementation's debugger, the traditional error-checking mechanisms provide sufficient debugging power for the average application. These ten constructs are described here as well as in Steele.

BREAK

This function is a method of gaining direct access to the debugger. The goal of this function is to freeze the dynamic behavior of your program so that you can explore why it behaves as it does. It is not intended to formally signal errors or to provide any recovery strategy.

Whenever you need to examine a particular executing part of your program, you can insert an invocation to this function, or set a *breakpoint*, at that spot. When the program reaches this location, it will immediately enter the debugger. You should be able to resume executing your program from the break once you have finished your examination. If you wish, you can supply a message that signals the breakpoint before the code enters the debugger. If you have entered several breakpoints into your program, it is a good idea to use these messages to distinguish among them.

WARN

Use this function to print a warning message when your program encounters an undesirable yet not fatal condition. Do not use a warning to advertise conditions that will undermine the execution of your program. Instead, use warnings to alert the user of some

abnormality that can be ignored for the time being but that should be corrected when execution is complete. Keep the message terse, informative, and grammatically correct.

For example, you could add warning code to your loader function to remind you to add an `in-package` statement to your files:

```
(defun loader (filename-or-stream)
  ...
  (unless (and (consp first-form)
               (eq (car first-form) 'in-package))
    (warn "File ~S does not begin with IN-PACKAGE.
           Loading into package ~S"
          (enough-namestring stream)
          (package-name *package*)))
  ...)
```

The next time you load a file that does not begin with an `in-package` expression, you receive the following warning:

```
> (loader "/mmiller/lisp/addup")
;;; Loading source file "/mmiller/lisp/addup.lisp"
;;; Warning: File "/mmiller/lisp/addup.lisp" does not begin with
;;;          IN-PACKAGE.  Loading into package "USER"
#P"//apollo13/mmiller/lisp/addup.lisp"
```

You can require `warn` to enter the debugger when it locates an error condition by setting the variable `*break-on-warnings*`. In this case, `warn` behaves like `break`, yet its intent is different; you use `break` in a specific debugging session, while you weave `warn` into the fabric of the program.

ERROR

The function `error` signals errors for which there is no dynamic recovery strategy. It then enters the debugger so that you can examine the current state of the stack to better understand the problem. The signaling message should explain the offense and identify the offending object. The debugger does not provide options for proceeding with operations for which `error` is invoked. However, there should be an option for returning to the top level of Lisp.

CERROR

The function `cerror` signals an error, enters the debugger, and provides a message that describes the action that you must take to continue with the execution. This function is a good candidate for situations where reasonable assumptions can be made about the input

or where default values can be substituted. It is also a good tool for verifying user input. If the user supplies the wrong type of data, `cerror` should signal an error, and the code following the call to `cerror` should take the necessary steps either to provide a quick fix or to solicit new data. It is good practice to check the new input for validity and to call `cerror` repeatedly until reasonable data are available. The error message should clearly explain the problem, and the options for continuation should succinctly and explicitly describe the correction that will be taken.

ASSERT

The macro `assert` signals an error when its first argument, which is a Lisp expression, evaluates to `nil`. It then allows the user to supply new argument values for reevaluation of the expression. It continues accepting new input and retesting until the given expression has a non-`nil` value. The error message should explain the cause of the error so that users can determine how to supply correct input.

Like `cerror`, this macro is good for verifying user input. However, it is more structured than `cerror` in that it not only signals the error but contains the mechanism for correcting the error—the Lisp expression that must succeed and `setf`-able variables that can be used in that expression. This macro automatically continues to prompt for the new value until one is supplied that satisfies the test. In contrast, when you use `cerror`, you must provide additional code to handle the correction and build the looping mechanism yourself.

CHECK-TYPE

The macro `check-type` verifies the data type of its first argument and signals an error if the argument is not of the correct type. The error message is generated automatically, but you can supply a string that specifies the type if you want the message to be more specific. Begin the string with the article *a* or *an* as appropriate, so that it can fit seamlessly into the predefined error message string.

Once in the debugger, the user can supply an argument of the correct type. The operation remains in the debugger until the correct data type is supplied:

```
> (defun type-test (test-arg)
    (check-type test-arg (integer) "an integer")
    (format t "The argument ~A is an integer." test-arg))
TYPE-TEST
> (type-test 4)
The argument 4 is an integer.
NIL
```

```
> (type-test "molly")
>>Error: The value of TEST-ARG, "molly", should be an integer

TYPE-TEST:
Original code: (NAMED-LAMBDA TYPE-TEST (TEST-ARG)
(BLOCK TYPE-TEST (CHECK-TYPE TEST-ARG # "an integer")
(FORMAT T "The argument ~A is an integer.")))
   Required arg 0 (TEST-ARG): "molly"
:C  0: Use a new value
:A  1: Abort to Lisp Top Level

-> :c
Use a new value
Value to use instead: .25
>>Error: The value of TEST-ARG, 0.25, should be an integer

TYPE-TEST:
Original code: (NAMED-LAMBDA TYPE-TEST (TEST-ARG)
(BLOCK TYPE-TEST (CHECK-TYPE TEST-ARG # "an integer")
(FORMAT T "The argument ~A is an integer.")))
   Required arg 0 (TEST-ARG): "molly"
:C  0: Use a new value
:A  1: Abort to Lisp Top Level

-> :c
Use a new value
Value to use instead: (+ 5 6)
Value is 11, OK? (Y or N): y
The argument 11 is an integer.
NIL
```

Note that `check-type` evaluated the user-supplied `test-arg` argument `(+ 5 6)` to be the integer 11.

Check argument types at the user-visible interface to an operation. Once data have passed through the interface, the receiving functions can then assume that they are receiving data of the correct type.

ETYPECASE, ECASE

The macros `etypecase` and `ecase` are error-checking equivalents of `typecase` and `case`, respectively. Neither of these conditional operators allows an `otherwise` clause; instead, if none of the keys is satisfied, an error is signaled. You cannot continue from errors signaled by these macros, but you can examine the stack from within the debugger. The error messages are predefined by the implementation. If you need to supply your

own error messages, use `typecase` or `case` and call `error` from within the `otherwise` clause.

CTYPECASE, CCASE

The macros `ctypecase` and `ccase` are error-checking equivalents of `typecase` and `case`, respectively; they signal continuable errors. Neither of these conditional operators allows an `otherwise` clause; instead, if none of the keys is satisfied, an error is signaled, and the debugger presents the options for continuing. The error messages are predefined by the implementation. If you need to supply your own error messages, use `typecase` or `case` and call `cerror` from within the `otherwise` clause.

These error constructs are flexible enough to cover a wide range of problems and are thus sufficient for the average program. However, for complex programs that require greater control over the types of errors that are signaled and the action taken in response to errors, Common Lisp provides the error-handling facility.

6.2.2.2 THE ERROR-HANDLING FACILITY

The error-handling facility is a latecomer to Common Lisp. It lets you define data structures called *conditions* to model errors and any other situations that affect the correct execution of your program and thus require attention. Like classes in CLOS, each condition is a new data type, and all predefined conditions form a type hierarchy. When one of these conditions arises during the execution of a program, a signaler mechanism checks to see if any functions have been defined to take action for conditions of this type. The signaler mechanism works much like generic dispatch in CLOS. A function that is invoked by the signaler is called a *handler* for the condition. Handlers can be thought of as dynamic methods. The handler can choose from a variety of actions, such as entering the debugger, signaling another condition, or throwing to a form called a *restart*; it can even decline to take any action.

For one-of-a-kind errors, such as anomalous behavior, the error-handling facility is not necessary. However, if you find that your program calls `error` often, you might want to define conditions to classify your errors. You then can define handlers to take actions appropriate to the different situations.

In the *Personal Planner*, we use a construct from the error-handling facility to check input in the user top-level function:

```
(defun personal-planner ()
  ;; All interaction with Personal Planner happens in the PLAN package.
  (let ((*package* (find-package 'plan)))
```

```
;; This loads the previous version of the database.
(restore-planner-database t)
;; This CATCH is just here for the QUIT command to THROW to.
(catch 'personal-planner
  (format t "~&Welcome to Personal Planner~%")
  (loop
    ;; This catches aborts from the debugger.
    (with-simple-restart
      (abort "Abort to Personal Planner Top Level")
    (loop
      (evaluate-command (read-command))))
  (format t "~&Back to Personal Planner Top Level~2%"))))
```

If the user enters erroneous data, the restart returns to the top level of the *Personal Planner* instead of forcing a return to Lisp top level from the debugger:

```
> (personal-planner)
Loading database...done
Welcome to Personal Planner
Command (? for options): ?
HELP, H, ADD, A, FIND, F, CHANGE, C, DELETE, D,
RENAME, R, QUIT, Q, NOSAVE, N, LISP, L.
Command (? for options): #.foo
>>Error: The symbol FOO has no global value

SYMBOL-VALUE:
   Required arg 0 (S): FOO
:C  0: Try evaluating FOO again
:A  1: Abort to Personal Planner Top Level
    2: Abort to Lisp Top Level

-> :a
Abort to Personal Planner Top Level
Back to Personal Planner Top Level
```

The macros `assert` and `check-type` and the functions `cerror`, `error`, and `warn` have been upgraded to work within the error-handling facility. These modifications allow greater flexibility within the debugger because you can specialize the behavior by condition type.

The error-handling facility introduces a number of new concepts and constructs to Common Lisp. A complete description of its impact is beyond the scope of this book. For information about these constructs and their role in the error-handling facility, see Steele.

6.3 DEBUGGING METHODOLOGY

Once you are familiar with the tools Lisp provides for debugging, you can develop a strategy for finding and fixing bugs. The strategy you choose might depend on the type of program you are debugging and the severity of the problem, but a good strategy embodies these elements:

- An understanding of what should happen
- An organized search for the first occurrence of the problem
- An analysis of previous mistakes to help locate other weak areas in the program

The following sections explore these aspects of the debugging process.

6.3.1 Understanding What Should Happen

It might seem trifling to say that you must understand what a program is supposed to do before you can determine what is going wrong, but gaining an understanding of the problem is as important to debugging as it is in formulating a program plan. You need a model of the program that you can compare to the results you are seeing. The relative difficulty of this step depends on which of two types of programs you are debugging:

- A program that you have written
- A program that was written by someone else

If you are trying to correct problems in your own program, you have the advantage of debugging module by module as you develop the code. When you reach a serious stumbling block, review the program plan. Have you followed your plan? If not, it is likely that the problem is related to the diversions from the plan. Trace those areas of the program that deviate from the plan to see how they behave. Pretend that you are the computer, and decide how you would react to the code in question. If you cannot get enough information from a trace, try stepping the code or setting a breakpoint and examining the stack. From a breakpoint, use the debugger to run a backtrace to see how the program control ended up here, and examine the values of local variables along the way. If looking deeper supports your intuition that you should have stuck to your plan, revise your code to follow the spec. If, on the other hand, you find that the variance is not the problem and that your program benefits from this change, revise your plan so that it accurately reflects the state of your program. Then make certain that the rest of the implementation is built upon the correct assumptions; that is, modify the rest of the program so that it behaves consistently with the updated algorithm.

When your program conforms to the plan, review the behavior at key points in the program's execution. Then use this information when finding the source of the anomaly by following the steps described in Section 6.3.2.

If you are debugging someone else's program, it is sometimes not as easy to determine what should be happening. Begin by reading the program documentation (if any exists) and by talking to the author of the program (if you have access to this person). If the program runs, set some breakpoints so that you can examine the stack and the values of local variables at key locations. Do all the function calls and values look reasonable? If not, why not? If running the program ends up in the debugger, do a backtrace and then examine the code for the functions that were called. Play computer, and formulate your results to the code. Run the program several times using different input data until you can predict its behavior with a fair amount of accuracy.

If the program is short, read the code; if it is quite large, read enough sample portions to gain a sense of the author's programming style and technique. Next, review the edit history of the program. Look for changes that might have an impact on the behavior you are seeing. Are any of the changes related; that is, are there a number of changes concentrated in one module of the program? If so, you may have located a weak spot of the program's algorithm; read the code in this module carefully.

6.3.2 Locating the Source of the Errors

Once you understand how the program should be working, you can begin your search for the errors. In this step, the Lisp debugging tools play a featured role, along with a somewhat antiquated yet effective set of tools—a pencil and a pad of paper. Use the pencil and paper to take notes on your strategy.and to catalog little pieces of evidence that might not make complete sense until more information is available. A necessary part of successful debugging is the ability to make decisions based on partial information.

The difficulty of locating where a problem first occurs in a program is relative to the type of error that is occurring—a run-time error that throws you into the debugger, a nontermination error, or the wrong result.

If the error lands you in the debugger, you have a vantage point from which to begin your search. Your first clue to the problem is the error message. Read it carefully; do not assume you know what the message says without reading it. The information supplied by the message is the system's best guess at the problem. Pay attention to the ways in which error messages are differentiated, and try to understand what types of problems cause which class of error.

Often, you can diagnose the problem from simply reading the error message, such as when you have misspelled a function or variable name and the message calls out that sym-

bol as being undefined. If you are not lucky enough to place the blame on a typo, backtrace from the current location; make a note (with your pad and pencil) of any function whose presence in the backtrace surprises you. Look at the values of local variables and write down any that do not look right. Try supplying new values and see what happens. Simplify your data, and run the program again. If the results are the same and you cannot figure out the error from the information provided at the location where the program entered the debugger, it is time to look elsewhere.

To find the problem area quickly, use a divide-and-conquer search technique to prevent you from wasting time looking for errors in the wrong places. Set a breakpoint at the last location where you think the program was executing correctly, and verify that everything works as expected at that point. Next set a breakpoint halfway between that point and the point that enters the debugger. How do things look there? Repeat this process until you have a narrow window of code that could be the culprit. Then, with your pencil and paper ready, hand-compute the operations and record your results. Next, trace or step through the code and compare the results with your handwritten ones. By continually narrowing the field, you can force the culprit out of its hiding place.

If you are receiving the wrong result or your program never terminates, the task of isolating the error point is more difficult. One time-tested technique for homing in on the problem is to insert print statements into your code that announce the values bound to certain variables at meaningful intervals during the program's execution. Then, using simplified data, rerun the program and examine the resulting transcript. Rule out those areas that return proper values, and concentrate on the area that immediately precedes the first spurious result. Finally, use the divide-and-conquer search procedure just described.

Once Lisp code is compiled, it loses some environment information that is useful for debugging. If you are debugging code that has been compiled in your image, check the optimization settings for your compiler. If the value of the `safety` setting is low, where zero is the lowest possible value, increase the safety level and recompile the code before debugging. Alternatively, load your source files and debug from interpreted code.

6.3.3 Learning the Value of Skepticism

If you have followed the preceding techniques and your program still does not work, consider these points:

- Do not assume that any part of the program works without checking it expressly. It is easy to overlook problems in code that you are confident of, and sometimes that confidence can be misplaced.

- When you have checked everything and are convinced every function works correctly, recheck those things that are easiest to overlook. Make certain that every function invocation specifies the function you expect. Are you invoking `define-x-coordinate` when you mean `define-y-coordinate`, or `viewport-screen-region` instead of `viewport-bitmap-region`?
- Look at the arguments you are supplying to functions. Are any arguments accidentally reversed? Are you specifying all the values you can, or are you relying on default values? Are the defaults correct for you?
- If you are using `format` to add print statements for debugging purposes, make certain that the `nil` result returned by `format` does not cause hidden side effects within the program being debugged.
- Know your own quirks. For example, make mental notes if you find you are often changing `compile` to `compile-file`. Then use the search command provided by your editor to look for and correct these idiosyncratic errors.

As a last resort, do not eliminate the possibility that you have uncovered a bug in your version of Common Lisp. If you think you have found an implementation bug, try to formulate the simplest test case that demonstrates the problem. The test case should show the problem in a context that you are certain should work without error. Then report the bug to your Common Lisp vendor, along with the test case; include information about the environment in which you ran the test case, such as the compiler optimization and memory management settings. It is a good idea to run the test case in a fresh Lisp image and to submit a transcript of the session.

6.3.4 Learning from Mistakes

Debugging can be a tedious and frustrating process, so it is important not to waste time and energy. If your program has built up a fair amount of state space before it started producing errors, resist the temptation to abort from the debugger. When you enter the debugger, use the available commands to nurse the program forward as far as possible. Do not let the first bug you find obscure the larger problem, and resist the urge to recompile and rerun a program after each individual bug fix. Instead, take notes on the behavior of the program beyond the first bug to determine the seriousness and scope of the error. Supply new values to the program from within the debugger to see how the behavior changes. If you can get your program back onto its normal execution path from within the debugger, take notes about what you have done and make certain to fix the source code later. Of course, if the

problem is a typo and you have just begun to run the program, there is no harm in aborting out of the debugger to edit and reload the code.

Develop an error threshold. If you have written a buggy program, there should be some point where fixing is not enough and rewriting is required. If the problem is in the algorithm you have chosen, rather than in its implementation, you have a lot of work to do, since you cannot simply patch a bad algorithm without serious harm to the overall organization of the program. Instead, once you have decided upon the correct algorithm, revise your program plan and identify what parts of the existing program must be rewritten to conform to the new algorithm. Then throw out the bad code and replace it with the new material. Rewrite as many sections as are necessary to seamlessly integrate the new algorithm.

Finally, remember that debugging is a seductive procedure, and it is easy to work for hours with little demonstrable progress. Admit when you have reached the point of exhaustion, and then quit for the night. Otherwise, you run the risk of confusing an already difficult situation by introducing careless errors or by proceeding down a fruitless path chosen out of desperation. Allow yourself a retreat from the problem; it usually seems less ominous when you look at it refreshed.

6.4 PROGRAM MAINTENANCE

It seems that no program is ever finished. As soon as everything is written and running well, someone will want to add a new feature or two. Over the life of a program, many modifications will be made, either by the program's author or by someone previously unfamiliar with the code.

Whenever you make changes to someone else's code, document the changes carefully, both in the edit history of the program and at the location of the changes. Include in the notes any assumptions you might have made in formulating the fix. If possible, try to adapt to the existing coding style. Then test the revised code thoroughly by running the program with old data and comparing the results with previous runs.

If someone wants you to add new features to a program that you have written, try to determine whether the change is compatible with the underlying plan for the program. Add the change to the program plan to see if it can be integrated successfully into the current algorithm. If it cannot, you have three choices—either rule out the change, rewrite the program using an algorithm that works with the change, or undermine the robustness of the program by allowing an ill-advised addition.

Chapter 7

EFFICIENCY

7.1 INTRODUCTION

There are two important types of efficiency—program efficiency and programmer efficiency. As hardware performance has increased, the cost of memory has decreased, and the value of good software has increased, programmer efficiency has become more important. Luckily, good style habits contribute to improving both types of efficiency. You should never have to sacrifice clarity for the sake of efficiency.

In this chapter, we divide the discussion of efficiency into the following sections:

- Section 7.2, Programming Efficiently, presents some general guidelines for writing efficient programs. You are familiar with many of these by now, since they are simply aspects of good programming style that we have been describing throughout this book.
- Section 7.3, Declarations and Compilation, describes the different Common Lisp declarations that help Lisp compilers to optimize code. It also covers the special form `eval-when`, which lets you control when code is evaluated.
- Section 7.4, Generalizing Program Code, weighs the costs and benefits of making program code more general.

We have placed this chapter last because it discusses aspects of efficiency that you should consider last in a programming project. You cannot postpone all considerations of efficiency until a program is complete, however. Good algorithms, suitable data structures, and readable code contribute to a program's efficiency throughout the development cycle.

7.2 PROGRAMMING EFFICIENTLY

Gaining efficiency is an ongoing process that begins with choosing a suitable algorithm. If the algorithm on which your program is based does not adequately address the problems

at hand, no amount of fine-tuning can make the program perform optimally. Similarly, whenever you have to wrestle with your code to make it fit your algorithmic model, you are bound to lose efficiency, either through poor readability, poor performance, or both. Thus, efficiency begins with a good approach to the problem.

The inherent efficiency of a good algorithm can suffer if the program is poorly organized. Locality of reference improves both the readability of a program as well as its paging performance. Thus, your program benefits from the time you spend in planning the organization. Get a clear understanding of the steps that the program must take to solve your problem, and sketch them out. Then follow those steps to completion. Flush out the pieces of the program independently, and write clear interfaces to them.

Avoid the temptation to redesign on the fly for the sake of making your code faster. You will have a greater chance of finding any problem areas once you have implemented the entire algorithm and can monitor the performance of the program as a whole.

Choosing appropriate data structures contributes to the efficiency of a program. A data structure should fit the data it represents and the operations being performed on those data. If you need to store large amounts of data and access them frequently, make certain that your data structures support efficient search and access mechanisms. Write your code in such a way that you can change the representation easily to accommodate different amounts or types of data.

Use the full expressive power of Lisp when writing code. The functionality provided by Common Lisp is broad enough to cover many types of situations, but it is also somewhat intimidating. It is sometimes more comforting to use only those constructs that you understand well rather than to explore other possibilities within the language; however, this tactic can result in using the wrong tool or data structure, that is, one that is not natural for the task or for the information you are manipulating. For a complete description of Common Lisp, see Steele.

Avoid writing Lisp programs as thinly masked translations of programs written in another language, such as C. This problem is common in introductory Lisp programming classes. Students who are familiar with C code soon realize that they can mimic C with Lisp, and so they use techniques suited to imperative languages rather than to Lisp. This practice is analogous to translating an English sentence to German by substituting the appropriate vocabulary words in place without recognizing the differences in syntax. Though the end result might be understandable to some degree, it is only barely related to actually speaking German.

When you have a clearly written program that performs the task it is expected to perform, you can begin to think about ways of making it more efficient. The first method of improving the efficiency of code is to insert declarations and to compile your code; this

method is discussed in detail in Section 7.3. The messages that the compiler produces might also reveal hidden deficiencies in the code, such as variables that are introduced yet not used. If you have chosen a good algorithm and data structures, have followed the algorithm well, and have payed attention to the readability of your code all along, adding declarations and compiling your code will most likely cover any remaining efficiency issues of your program.

However, if adding declarations does not improve the performance of your code, write some simple diagnostic tests to find the exact location of the sluggishness, and then examine that location carefully. Common Lisp provides the macro `time` and the function `room` for gathering implementation-dependent information about the time and memory resources your program requires. Many implementations also provide more sophisticated facilities for monitoring performance. It is very important not to rewrite code with a mind to improving efficiency without testing and measuring first. Giving in to the temptation to rewrite before testing has wasted countless programmer hours.

When you have pinpointed the time sinks in your program, look for the problem spots described in the following sections. Most of all, remember that premature optimization is the root of most bugs. Sketch your program out, test it, revise as you verify that the design is usable and correct, and then selectively optimize as you proceed and find where the real cost lies.

7.2.1 Compounded Problems

Inefficiencies in code that is executed repeatedly, such as iterative loops and recursive functions, can be very expensive, since the problems they manifest happen over and over. Look carefully at the interior of such code. Are you binding special variables? Remember that Lisp must keep track of the previous values of those variables each time they are bound. Is storage being created unnecessarily? Excessive consing can cause frequent garbage collection and increased paging activity, which can significantly slow down the execution of your program. Can you reduce the number of function calls or move any functions being called to the outside of the code that is repeatedly executed?

Sometimes you might need to modify your algorithm to make a loop more efficient. For example, we could reduce the amount of storage required for the `loop` version of the function `generate-normalized-random-list` described in Chapter 3 by relying on destructive modification and by changing data structures. As you will recall, the `loop` example in Section 3.3.12 collected a list of random numbers and then used that list as an iteration control construct for building a new, normalized list of numbers; the first list was discarded in the process. In the following version of `generate-normalized-random-list`, the list of denormalized numbers that is collected in the `loop` body is destructively

modified during the normalization step in the `loop` epilogue:

```
(defun generate-normalized-random-list (n max-value)
  (loop repeat n
        for value = (random max-value)
        sum value into total
        collect value into result
        finally
        (return (loop for tail on result
                      with average = (/ total n)
                      do (setf (car tail) (- (car tail) average))
                      finally (return result)))))
```

By reusing the list `result` to hold the normalized numbers, we have saved half the storage of our first `loop` version.

Next we use an array instead of a list to hold the denormalized numbers; the same array is destructively modified to hold the normalized numbers. This version has an additional storage savings, since in most implementations simple arrays take about half the space as lists with the same number of elements:

```
(defun generate-normalized-random-list (n max-value)
  (loop with result = (make-array n)
        for i from 0 below n
        as value = (random max-value)
        do (setf (svref result i) value)
        sum value into total
        finally
        (return (loop for i from 0 below n
                      with average = (/ total n)
                      do (setf (svref result i)
                               (- (svref result i) average))
                      finally (return result)))))
```

This solution becomes more attractive as there are more numbers to be normalized.

7.2.2 Data Manipulation Problems

Other problems might not be as costly as those described in the previous section, but they can still impair otherwise good code. The following guidelines might help you prevent some unnecessary efficiency loss:

- Examine the way multiple values are handled. Use Common Lisp constructs for passing multiple values instead of creating unnecessary lists.

- Look for searching and sorting operations in the problem area. Use the most efficient techniques for the types of data involved, and make certain that the predicate that controls the sort has been compiled.
- Check the arguments to, and the results of, arithmetic operations. Prevent your program from coercing numerical results into the correct format by using the same format for all numerical arguments whenever possible. Doing so also makes expressions easier to read.
- When constructing or comparing data structures, use `:test` arguments that most efficiently compare the data that are stored.
- See how often your program reads or writes data. Make certain that all of your I/O operations are necessary.

7.3 DECLARATIONS AND COMPILATION

Common Lisp allows you to run your program in an interpreted form as well as in a compiled form. The compiled form runs faster than the interpreted form because it uses instructions that can be directly executed at the machine level. Also, Lisp knows that the compiled version of a program consists of a finite set of constructs, whereas in the interactive interpreted environment, Lisp must be prepared to understand an exhaustive set of constructs. You can improve the performance of your compiled code by adding certain *declarations*, which give the compiler information from which it might produce more efficient code. Declarations also provide more information to human readers of your program because they serve to document the intended types of objects and operations.

You can use the special form `declare` to add declarations to your code. You can only add declarations to certain macros and special forms, but these include `defun`, `defmacro`, `locally`, `let`, `do`, `flet`, `labels`, `with-open-file`, and `multiple-value-bind`, so they cover a large territory. (See Steele for complete information about `declare` and the declaration specifiers allowed in Common Lisp.) Except for `special` declarations, which specify global variables, declarations cannot change the meaning of a program, and their use is optional. Since getting the best performance out of your code is hardly ever optional, add declarations whenever the compiler might be able to use them effectively.

The `type` declaration is probably the most important of all declarations. Common Lisp data objects are typed objects; that is, each data object carries information about its type. Many Lisp constructs that manipulate Lisp data objects must first check the type of the object before proceeding with the rest of the operation, a process called *type checking*. Since type checking occurs with every Lisp object, it can incur a large amount of overhead. (Lisp machines handle type checking in microcode and hardware, so type declarations

are not necessary in Lisp-machine code.) Any time you declare the type of an object, the compiler can avoid the type-checking step. In some cases, the compiler can also use specially optimized type-specific operations that increase the speed at which that object is accessed.

You can use the `ftype` declaration to indicate to the compiler that a function can only accept arguments of certain types and will only produce results of certain types.

The `the` special form provides a method for declaring the types that result when an expression is evaluated. By wrapping a `the` construct around an expression, you indicate to the compiler that the value returned by the expression will always be of a certain type. You can combine `the` with the function `values` when the evaluated expression produces more than one type; in this case, you can declare as many types as there are values returned.

By combining `type` declarations with `the` declarations, you can inform the compiler that both the arguments to a function and the results returned by a function will be of certain types. If you are performing arithmetic operations and you have declared the arguments and the results to be of type `fixnum`, many compilers can perform special optimizations that result in very fast code.

Though type declarations are very important, other declarations can improve the efficiency of your code as well. An `inline` declaration or proclamation tells the compiler to replace the specified Lisp function call directly with compiled code for that function; this process is called *open coding*. Many compilers automatically open-code certain Common Lisp constructs, such as `car` and `cdr`, to improve efficiency. Open coding is useful in situations where the overhead associated with invoking a function is greater than the overhead associated with executing that function. However, its benefits are not without cost:

- If you compile a call to a function that is proclaimed `inline` and you redefine the function later, you must recompile the function as well as any other functions or macros that use it in order for the definition to be effective throughout the environment.
- Open coding can also bloat the compiled image and thus should be used judiciously. See Section 7.3.2 for information about saving space with open coding.

The `optimize` declaration gives the compiler a general view of what qualities you are concerned about in compiled code. The qualities you can specify are `speed`, `safety`, `compilation-speed`, and `space`. You indicate the importance of these qualities in your code by giving them a value from 0 to 3, where 3 is the highest value and 0 the lowest. If you are concerned about how fast the compiled code will run, specify a high value for `speed`. If you are concerned about the robustness of the code and want to make certain that all possible error checks occur, specify a high value for `safety`. If you want to be

able to compile your code quickly, increase the value of `compilation-speed`. If you want your compiled code to be as small as possible, emphasize `space`.

In most Lisp implementations, as your compiled code is optimized for speed, it loses debugging information that was available in the interpreted environment. Thus, it is best to debug your program in interpreted form before compiling it for speed. Compile first while optimized for safety to retain debugging information. Then incrementally shift the emphasis to speed as you become more confident in the robustness of your code.

7.3.1 An Example

The following example shows how `type` declarations work. Assume the file `test.lisp` contains the following function definitions:

```
(in-package "USER")

(proclaim '(optimize (speed 3) (safety 1)
                     (space 0) (compilation-speed 0)))

;;; Compare simple strings for equality.
(defun compare-simple-strings
    (string1 string2 start1 end1 start2 end2)
  (do ((i1 start1 (1+ i1))
       (i2 start2 (1+ i2)))
      ((or (= i1 end1) (= i2 end2))
       (values (and (= i1 end1) (= i2 end2)) i1 i2))
    (unless (char= (aref string1 i1) (aref string2 i2))
      (return (values nil i1 i2)))))

;;; Compare simple strings for equality, and use declarations
;;; for efficiency.
(defun fast-compare-simple-strings
    (string1 string2 start1 end1 start2 end 2)
  (declare (fixnum start1 end1 start2 end2))
  (declare (simple-string string1 string2))
  (do ((i1 start1 (1+ i1))
       (i2 start2 (1+ i2)))
      ((or (= i1 end1) (= i2 end2))
       (values (and (= i1 end1) (= i2 end2)) i1 i2))
  (declare (fixnum i1 i2))
  (unless (char= (aref string1 i1) (aref string2 i2))
    (return (values nil i1 i2)))))
```

```
;;; Time several iterations of invoking the specified function with
;;; the arguments supplied.
(defun test (iterations function str1 str2 start1 start2 end1 end2)
    (time (dotimes (i iterations)
            (funcall function str1 str2 start1 start2 end1 end2))))
```

The difference between `compare-simple-strings` and `fast-compare-simple-strings` is that the latter definition contains `type` declarations.[1] We can see the effects of this difference by compiling and loading the file and then timing iterative invocations of the two functions. Our compiler settings emphasize run-time speed over safety and compilation speed:

```
> (load (compile-file "/mmiller/programmers-guide/work/test"))
;;; You are using the compiler in production mode
;;; (compilation-speed = 0)
;;; Generation of argument count checking code is enabled
;;; (safety = 1)
;;; Optimization of tail calls is enabled (speed = 3)
;;; Reading source file "/mmiller/programmers-guide/work/test.lisp"
;;; Writing binary file "/mmiller/programmers-guide/work/test.lbin"
;;; Loading binary file "/mmiller/programmers-guide/work/test.lbin"
#P"//apollo13/mmiller/programmers-guide/work/test.lbin"

;; We ran a few preliminary tests first, so that all the
;; information could be paged in.  Then we ran these tests.
> (test 25000 #'compare-simple-strings "molly" "llama" 2 3 1 2)
Elapsed Real Time = 4.283896 seconds
Machine Run Time = 4.090856 seconds
NIL
> (test 25000 #'fast-compare-simple-strings "molly" "llama" 2 3 1 2)
Elapsed Real Time = 1.944836 seconds
Machine Run Time = 1.602968 seconds
NIL
```

The optimized version shows more than a 50 percent speedup at 25,000 iterations.

These results demonstrate the difference that declarations make in the run-time efficiency of a single function. Imagine the difference if optimizations were added to every bottleneck in your program.

1. The symbol `type` is not necessary in these declarations, since the types being declared are Common Lisp atomic types.

7.3.2 Evaluation Control

Common Lisp provides the special form `eval-when` to allow you to control the situations when code is evaluated. There are three situations in which code can be executed:

- `eval`—in an interpreted environment
- `compile`—during compilation
- `load`—when compiled code is loaded

In Lisp, you can perform arbitrary operations during macro expansion; thus, at compile time, you may need objects that are only useful during macro expansion. For this reason, Lisp includes `eval-when`. Such a construct is not necessary in a language like C, which has a restricted macro language. C simply does not allow you to compute arbitrary C programs and then use the result in the expansion of macros. Note that `eval-when` is meant to be used at Lisp top level. It often appears at the top of program files, though it might be used inside a `progn` or in a macro that expands into a top-level construct.

For top-level forms and `defun` function definitions, code is evaluated and executed in the interpreted environment or when it is loaded after compilation; that is, the default behavior is

```
(eval-when (eval load) ...)
```

Sometimes you need to make Lisp objects available only in the compilation environment; you do not want them to be loaded into the resulting binary file. For example, suppose you had a macro that called a helper function during macro expansion, and the macro was in the same file in which it was used. The helper function was going to help construct the form that the macro was going to return, but the helper function was not intended to be part of the interface. For this arrangement to work, the function must be defined at compile time when the macro is expanded. You can ensure this situation by wrapping the helper function inside an `eval-when` construct, as the following contrived example shows:

```
(eval-when (eval compile)

;;; MAKE-FROBBOZ-MAKER constructs a call to MAKE-FROBBOZ.  It does
;;; something interesting with the options given.
(defun make-frobboz-maker (&rest options)
  `(make-frobboz ,@(do-something-interesting-with options)))
```

```
;;; DEFFROBBOZ creates a global frobboz.
(defmacro deffrobboz (name &rest options)
  `(defvar ,name ,(make-frobboz-maker options)))

;;; WITH-FROBBOZ binds a local frobboz.
(defmacro with-frobboz ((name &rest options) &body body)
  `(let ((,name ,(make-frobboz-maker options)))
     ,@body))

)
```

The rest of the file uses `deffrobboz` and `with-frobboz`. When the compiled file is loaded, the definition for `make-frobboz-maker` is not needed. The invocation of the macro has been expanded during compilation; that is, its use has been compiled. The `eval-when` construct ensures that correct behavior occurs in an interpreted environment as well as at compile time. If `make-frobboz-maker` were not inside the `eval-when`, an error would be signaled when the file was compiled:

```
>>Error: Undefined function MAKE-FROBBOZ-MAKER
```

If you are using a macro simply as a building block of a program and the macro definition resides in the same file in which it is used, you can save the space of loading in that definition by specifying

```
(eval-when (eval compile) ...)
```

You can also use `eval-when` in this way around functions that are proclaimed `inline` to save space at load time.[2]

This technique will not work if you need to incrementally define a function in the file. In such a situation, you must use `(eval-when (eval compile load))` to keep the macro definitions available.

If you want top-level forms to be executed during compilation as well as in the run-time environment, you must specify

```
(eval-when (eval compile load) ...)
```

Macros defined with `defmacro` implicitly capture this type of evaluation behavior.

2. Or, you could put these compiler-dependent forms in separate files that can be compiled and loaded in advance of the files that use them. This treatment makes `eval-when` unnecessary (see Section 5.4).

7.4 GENERALIZING PROGRAM CODE

To some degree, generalizing a program makes it more usable and improves maintenance efficiency because extensions are built in. However, generalizations can make a program execute more slowly; they can also make the code more complex and thus more difficult to write and to maintain. For example, the first version of the entity database for the *Personal Planner* did not support entities of the same name, even if they were of two different types. A new entity was pushed onto a list so that it shadowed any entity in the database that had the same name:

```
;;; Install an entity by name.
(defun add-entity (name entity)
  (pushnew entity (gethash name *entity-table*)))
```

A revised version of the database accepted entities that have the same name as long as they were not the same type. However, the code for adding an entity to the database became more complicated in achieving this generality:

```
;;; Install an entity by name
(defun add-entity (name entity)
  (let* ((existing-entities (gethash name *entity-table*))
         (new-class (class-of entity)))
    (dolist (old existing-entities)
      (let ((existing-class (class-of old)))
        (when (or (subtypep new-class existing-class)
                  (subtypep existing-class new-class))
          (error 'conflicting-entities entity old))))
    (setf (gethash name *entity-table*)
          (cons entity existing-entities))))
```

We later generalized the entity database further (see Section 4.4).

The time spent in generalizing code must be weighed against the likelihood that the generalizations will become necessary. Unnecessary generalization simply clutters up your code. When writing a code facility, ask yourself how you might generalize it up one level. If you can find a modest revision that improves generality, it is probably worth the effort to generalize. If all the generalizations that come to mind are on a grand scale, sketch out in comments what directions more general code might take and reconsider the issue once you have a working model that addresses the specific issues of the current problem. If you can find no areas that can be made more general, be clear about your reasons for using special-purpose code, and explain those reasons completely in comments.

Also take care that your generalizations do not get in the way of how the user must specify data. Generalizations should be as seamless as possible to the user. If a generalized

control facility requires a variable number of arguments depending on the type of the operation, set up default values so that the user does not have to specify arguments that are not natural to the operation.

THE PERSONAL PLANNER

This appendix contains all the finished code for the *Personal Planner*. The code is shown in the order in which we load the program files. Note that, in all files except the top-level file `load.lisp`, we have omitted the file headers to save space and to avoid redundancy. Each file is identified by its file name. We show a sample interaction with the program in Appendix B.

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-
;;;
;;; Load or compile the files in Personal Planner
;;;
;;; Programmer: Eric Benson
;;;
;;; Edit-History:
;;;
;;; Created: Sun Jan 15 10:57:33 1989
;;;
;;; End-of-Edit-History

(in-package "USER")

;;; A simple mechanism for compiling and loading these files
;;; in the correct order.  The variables *PLANNER-SOURCE-
;;; PATHNAME-TYPE* and *PLANNER-BINARY-PATHNAME-TYPE* and
;;; the macro PLANNER-BATCH-COMPILATION-WRAPPER will have to be
;;; modified when this program is ported to a new Lisp
;;; implementation.

;;; There are relatively few compile/load order dependencies in
;;; these files.  Superclasses are always defined before their
```

```
;;; subclasses, in spite of the fact that this is not required
;;; by CLOS.

(defparameter *planner-files*
  '("package"          ;Setting up PLANNER package required by all files
    "plist"
    "extra-types"      ;Miscellaneous extra types used in later files
    "planner-object"   ;DEF-PLANNER-CLASS needed to compile many files
    "interned-mixin"
    "entity-object"
    "time-zone"        ;Needed before calling MAKE-STATE
    "state-object"     ;Needed before loading states
    "address-object"
    "phone-number-object"
    "city-object"
    "person-object"
    "organization-object"
    "event-object"
    "meeting-object"
    "date-constants"   ;Constants needed by DATE-WRITE and DATE-PARSE
    "date-write"
    "date-parse"
    "toplevel"         ;Command interpreter
    "type-dispatch"
    "accept"           ;Generalized input based on data type
    "edit"             ;Generalized object editing
    "display"          ;Generalized display
    "database"
    "file-constants"   ;Constants used for file saving and restoring
    "file-save"        ;Save any object to a file
    "file-restore"     ;Read any object from a file
  "A list of the files in Personal Planner.")

(defvar *planner-source-pathname-type*
  #+Lucid (first *load-source-pathname-types*)
  #-Lucid (error "Unknown source type")
  "The standard pathname type for Lisp source files
   on this machine.")

(defvar *planner-binary-pathname-type*
  #+Lucid (first *load-binary-pathname-types*)
  #-Lucid (error "Unknown binary type")
  "The standard pathname type for compiled Lisp files
   on this machine.")
```

```
(defmacro planner-batch-compilation-wrapper (&body body)
  "This macro gets wrapped around a sequence of calls to
   COMPILE-FILE so that per-file warnings, such as those
   about unknown functions, will be delayed until the end."
  (let ((system-dependent-wrapper
         #+Lucid 'with-deferred-warnings
         #-Lucid 'progn))
    `(,system-dependent-wrapper ,@body)))

(defun load-planner
    (&optional (action :binary)
               (directory *default-pathname-defaults*))
  "Load Personal Planner.  ACTION can be any of :BINARY, :SOURCE, or
   :COMPILE. If ACTION is :BINARY, load compiled files; if :SOURCE,
   load source files; if :COMPILE, compile the source files and load
   the resulting binaries. DIRECTORY is the location of the files."
  (let ((source-default
          (make-pathname :defaults directory
                         :type *planner-source-pathname-type*))
        (binary-default
          (make-pathname :defaults directory
                         :type *planner-binary-pathname-type*)))
    ;; Just load, or compile and then load, the files in the order given.
    (planner-batch-compilation-wrapper
      (dolist (file *planner-files*)
        (let ((source-file (merge-pathnames file source-default))
              (binary-file (merge-pathnames file binary-default)))
          (case action
            (:compile
             (compile-file source-file)
             (load binary-file))
            (:binary
             (load binary-file))
            (:source
             (load source-file)))))))
t)

;;;; file package.lisp
;;;; Create the PLANNER package for Personal Planner

(in-package "USER")

;;; The PLANNER package uses the LISP, LUCID-COMMON-LISP, and CLOS
;;; packages.  The LISP package contains the symbols documented in
```

```
;;; Steele, the LUCID-COMMON-LISP package contains the Lucid
;;; extensions to Common Lisp, and the CLOS package contains the CLOS
;;; constructs we need.  We use :IMPORT-FROM explicitly to avoid
;;; problems with CLOS::CLASS-PROTOTYPE.
(defpackage "PLANNER"
  (:use "LISP" "LCL" "CLOS")
  (:import-from "CLOS" "CLASS-PROTOTYPE"))
```

```
;;;; file plist.lisp
;;;; Define the PROPERTY-LIST-MIXIN class

(in-package "PLANNER")

;;; Instances of PROPERTY-LIST-MIXIN have the ability to store
;;; properties in the same manner as symbols.  Since GET and
;;; REMPROP are not generic functions, we have to define new
;;; functions for property list access to these objects.
;;; These new functions are GET-PROPERTY (analogous to GET),
;;; SET-PROPERTY (analogous to SETF of GET), and REMOVE-PROPERTY
;;; (analogous to REMPROP).

(defclass property-list-mixin ()
  ((property-list :type list :accessor property-list :initform '())))

(defmethod get-property ((object property-list-mixin) property)
  (getf (property-list object) property))

(defmethod set-property ((object property-list-mixin) property value)
  (setf (getf (property-list object) property) value))

(defmethod remove-property ((object property-list-mixin) property)
  (remf (property-list object) property))
```

```
;;;; file extra-types.lisp
;;;; Define various types used in Personal Planner

(in-package "PLANNER")

;;; These data types are used solely for the functions ACCEPT,
;;; DISPLAY, and EDIT.
```

```
;;; (LIST-OF anything) is equivalent to LIST but provides more
;;; information when desired, such as when prompting for input.
(deftype list-of (&optional type)
  (declare (ignore type))
  'list)

;;; MENU-MEMBER and TERSE-MEMBER are exactly equivalent to MEMBER.
;;; They are used only to indicate input style for ACCEPT.
(deftype menu-member (&rest elements)
  `(member ,@elements))

(deftype terse-member (&rest elements)
  `(member ,@elements))

;;; (OPTIONAL type) is the same as (OR type NULL).  When
;;; prompting for input, a y-or-n-p question is asked first.
(deftype optional (&optional type)
  `(or ,type null))

;;; INDEX is a non-negative integer
(deftype index () `(integer 0 *))

;;; UNIVERSAL-TIME and ELAPSED-TIME are both equivalent to INTEGER.
;;; ELAPSED-TIME is the number of seconds between two events.
;;; UNIVERSAL-TIME is the number of seconds since midnight,
;;; January 1, 1900, Greenwich Mean Time.
(deftype elapsed-time () 'integer)

(deftype universal-time () 'integer)



;;;; file planner-object.lisp
;;;; Define the basic PLANNER-OBJECT for Personal Planner

(in-package "PLANNER")

;;; BASIC-PLANNER-OBJECT is the parent of all classes defined in
;;; the Personal Planner.
(defclass basic-planner-object ()
  ())
```

```
;;; Events and entities both inherit from the PLANNER-OBJECT, which
;;; has the PROPERTY-LIST-MIXIN.  We may later add other general
;;; functionality to PLANNER-OBJECT.
(defclass planner-object (basic-planner-object property-list-mixin)
  ())

;;; DEF-ABSTRACT-PLANNER-CLASS and DEF-CONCRETE-PLANNER-CLASS take
;;; arguments just like DEFCLASS.  We save away some of the
;;; information for later use.

;;; DEF-PLANNER-CLASS is not used directly.  It's only defined for
;;; DEF-ABSTRACT-PLANNER-CLASS and DEF-CONCRETE-PLANNER-CLASS to
;;; expand into.
(defmacro def-planner-class
    (name abstract-p superclasses slot-specifiers
     &rest class-options)
  `(progn
     (defclass ,name ,superclasses
       ,(remove-planner-specific-slot-info slot-specifiers)
       ,@class-options)
     (save-planner-class-information
       ',name ',abstract-p ',superclasses ',slot-specifiers)))

(defun remove-planner-specific-slot-info (slot-specifiers)
  ;; Take out the slot specifiers that DEFCLASS doesn't use,
  ;; which is :DISPLAY right now.
  (loop for slot-spec in slot-specifiers
        collect
        (let ((copy (copy-list slot-spec)))
          (remf (cdr copy) :display)
          copy)))

;;; Abstract means this class will never be instantiated.  It is
;;; only used as a superclass.
(defmacro def-abstract-planner-class
    (name superclasses slot-specifiers &rest class-options)
  `(def-planner-class ,name t ,superclasses
                      ,slot-specifiers ,@class-options))

;;; Concrete means this class can be instantiated.
(defmacro def-concrete-planner-class
    (name superclasses slot-specifiers &rest class-options)
  `(def-planner-class ,name nil ,superclasses
                      ,slot-specifiers ,@class-options))
```

```
(defvar *planner-class-information* '())

(defstruct (planner-class-information (:conc-name planner-class-))
  name
  abstract-p
  superclasses
  slot-specifiers)

(defun save-planner-class-information
    (name abstract-p superclasses slot-specifiers)
  (delete name *planner-class-information*
          :test #'eq
          :key #'planner-class-name)
  (push (make-planner-class-information
          :name name
          :abstract-p abstract-p
          :superclasses superclasses
          :slot-specifiers slot-specifiers)
        *planner-class-information*)
  name)

;;;; Functional interface to planner class information
;;;; This interface is used by ACCEPT, DISPLAY, and EDIT.

(defun find-planner-class-information (type)
  (find type *planner-class-information* :key #'planner-class-name))

(defun get-slot-information (type)
  ;; Collect a list of all the slot information for this type.
  ;; This always goes from highest superclass to lowest, from left
  ;; to right, with duplicates ignored.
  (get-recursive-slot-information type '()))

(defun get-recursive-slot-information (type seen-so-far)
  (if (member type seen-so-far)
      (values '() seen-so-far)
      (let ((planner-class (find-planner-class-information type)))
        (if (not planner-class)
            (values '() seen-so-far)
            (labels
                ((get-superclass-slot-information
                     (superclasses collected-slots seen-so-far)
```

```
        (if (null superclasses)
            (values
              (append collected-slots
                      (planner-class-slot-specifiers
                        planner-class))
              (cons type seen-so-far))
            (multiple-value-bind
                (slots-so-far seen-so-far)
              (get-recursive-slot-information
                (first superclasses)
                seen-so-far)
              (get-superclass-slot-information
                (rest superclasses)
                slots-so-far
                seen-so-far)))))
      (get-superclass-slot-information
        (planner-class-superclasses planner-class)
        '()
        seen-so-far))))))

(defun slot-information-names (slot-information)
  (mapcar #'car slot-information))

(defun find-slot-description (slot-name slot-information)
  (assq slot-name slot-information))

(defun slot-description-name (slot-info)
  (car slot-info))

(defun slot-description-type (slot-info)
  ;; The default type is T.
  (or (getf (cdr slot-info) :type) 't))

(defun slot-description-initarg (slot-info)
  (getf (cdr slot-info) :initarg))

(defun slot-description-display (slot-info)
  (getf (cdr slot-info) :display))

;;; Return a list of all the instantiable classes.
(defun concrete-planner-classes ()
  (loop for class-info in *planner-class-information*
        unless (planner-class-abstract-p class-info)
        collect (planner-class-name class-info)))
```

```
;;;; file interned-mixin.lisp
;;;; Define the INTERNED-MIXIN class

(in-package "PLANNER")

(defclass interned-mixin ()
  ())

;;; An interned class has a shared slot that contains the table
;;; holding all the instances of that class.  Note that subclasses
;;; should NOT use DEF-INTERNED-CLASS; the slot is shared by all
;;; the subclasses.  The class defined with DEF-INTERNED-CLASS is
;;; the root of a set of classes all of whose instances are
;;;interned together.
(defmacro def-interned-class (name superclasses slot-specifiers
                                   &rest class-options)
  `(defclass ,name (interned-mixin ,@superclasses)
     ((intern-table :allocation :class
                    :initform (make-hash-table :test #'equalp)
                    )
      ,@slot-specifiers)
     ,@class-options))

;;; Some of the following functions use CLASS-PROTOTYPE.  This is
;;; part of the meta-object protocol.  For every class whose
;;; metaclass is STANDARD-CLASS, there is an instance of that
;;; class, called the prototype.  It is used here as a key
;;; for discrimination, to allow us to define methods before
;;; we have created an instance.

;;; FIND-INSTANCE takes initargs like MAKE-INSTANCE but returns
;;; NIL if the instance didn't already exist.
(defmethod find-instance ((type symbol) &rest initargs)
  ;; TYPE must be the name of a class.
  ;; This just passes the buck to a method defined for
  ;; the class given. The prototype is used for discrimination.
  (apply #'find-instance
         (class-prototype (find-class type))
         initargs))

(defmethod find-instance ((class standard-class) &rest initargs)
  ;; CLASS must have a method for FIND-INSTANCE defined.
  ;; The prototype is used for discrimination.
  (apply #'find-instance
         (class-prototype class)
         initargs))
```

```
(defmethod find-instance ((prototype interned-mixin) &rest initargs)
  (gethash (apply #'initargs-intern-key prototype initargs)
           (slot-value prototype 'intern-table)))

;;; Many callers of FIND-INSTANCE need to generate an error.
(defun find-instance-or-error (class &rest initargs)
  (or (apply #'find-instance class initargs)
      (error "No instance ~S ~{~S~^ ~}" class initargs)))

;;; INTERN-INSTANCE should be used to create objects of interned
;;; classes, not MAKE-INSTANCE.  It takes initargs just like
;;; MAKE-INSTANCE but will return a previously created instance
;;; if there is one in the table.
(defmethod intern-instance ((type symbol) &rest initargs)
  ;; Call the method on the prototype, just like in FIND-INSTANCE.
  (apply #'intern-instance
         (class-prototype (find-class type))
         initargs))

(defmethod intern-instance ((class standard-class) &rest initargs)
  ;; Call the method on the prototype, just like in FIND-INSTANCE.
  (apply #'intern-instance
         (class-prototype class)
         initargs))

(defmethod intern-instance ((prototype interned-mixin)
                            &rest initargs)
  (let ((table (slot-value prototype 'intern-table))
        (key (apply #'initargs-intern-key prototype initargs)))
    (or (gethash key table)
        (setf (gethash key table)
              (apply #'make-instance
                     (class-of prototype) initargs)))))

;;; REINTERN-INSTANCE gives another name to an existing instance.
(defmethod reintern-instance ((instance interned-mixin)
                              &rest initargs)
  (let* ((key (apply #'initargs-intern-key instance initargs))
         (table (slot-value instance 'intern-table))
         (existing-instance (gethash key table)))
    (if existing-instance
        (error "~S already exists for ~{~S~^ ~}"
               existing-instance initargs)
        (setf (gethash key table) instance))))
```

```
;;; UNINTERN-INSTANCE removes an existing name for an instance.
(defmethod unintern-instance ((instance interned-mixin)
                              &rest initargs)
  (remhash (apply #'initargs-intern-key instance initargs)
           (slot-value instance 'intern-table)))

;;; INITARGS-INTERN-KEY takes a prototype and initargs and returns
;;; the key for those initargs.  The prototype argument is used for
;;; discrimination.  The result is used as the key in an EQUALP hash
;;; table.  Thus, it only makes sense to return values for which
;;; EQUALP makes sense.  That includes symbols, strings, numbers,
;;; lists, and vectors of appropriate values.  It does not include
;;; instances or structures.
(defgeneric initargs-intern-key (prototype &rest initargs))

;;; MAP-INSTANCES calls a function on each of the instances in the table.
(defmethod map-instances ((type symbol) function)
  (map-instances (class-prototype (find-class type)) function))

(defmethod map-instances ((class standard-class) function)
  (map-instances (class-prototype class) function))

(defmethod map-instances ((prototype interned-mixin) function)
  (maphash
    #'(lambda (key value)
        (declare (ignore key))
        (funcall function value))
    (slot-value prototype 'intern-table)))



;;;; file entity-object.lisp
;;; Define the class ENTITY for Personal Planner

(in-package "PLANNER")

;;; Entities include locations, people, organizations, events.
;;; Every entity has a name and a list of nicknames.  Every
;;; entity is stored in the entity table under all its names.

;;; ENTITY is defined with DEF-INTERNED-CLASS; it is the top
;;; of a hierarchy of classes that are kept in a table by name.
(def-interned-class entity (planner-object)
  ((name :type string :accessor entity-name :initarg :name)
   (nicknames :type (list-of string) :initform '()
              :accessor entity-nicknames)))
```

```
;;; Save the class information for input and output functions.
(save-planner-class-information
  'entity t '()
  '((name :type string :initarg :name :display :terse)
    (nicknames :type (list-of string))))

(defmethod print-object ((entity entity) stream)
  (let ((name (if (slot-boundp entity 'name)
                  (entity-name entity)
                  "Anonymous")))
    (if *print-escape*
        (format stream
                "#<~:(~A~) ~S>"
                (class-name (class-of entity))
                name)
        (princ name stream))))

;;; All this does is make sure that the shared slot INTERN-TABLE is
;;; initialized in the class prototypes.
(make-instance 'entity)

;;; See the DEFGENERIC for INITARGS-INTERN-KEY for documentation.
(defmethod initargs-intern-key ((prototype entity) &key name)
  (list (class-name (class-of prototype))
        name))

;;; Add a new name for this instance.  If it has no name, make this
;;; the name; otherwise make this a nickname.
(defmethod reintern-instance :before ((entity entity) &key name)
  (if (slot-boundp entity 'name)
      (setf (entity-nicknames entity)
            (nconc (entity-nicknames entity) (list name)))
      (setf (entity-name entity) name)))

(defmethod unintern-instance :after ((entity entity) &key name)
  ;; If the name being uninterned is the entity-name, promote the
  ;; first nickname to the name slot.
  (let* ((nicknames (entity-nicknames entity))
         (new-nicknames
          (delete name nicknames :test #'string-equal)))
```

```
    (when (slot-boundp entity 'name)
      (let ((old-name (entity-name entity)))
        (when (string-equal name old-name)
            (unless (null new-nicknames)
              (setf (entity-name entity) (first new-nicknames))
              (setq new-nicknames (rest new-nicknames))))))
    (setf (entity-nicknames entity) new-nicknames)))

;;; A locatable-entity is one that has an address and phone number
;;; (such as a person or organization).
(def-abstract-planner-class locatable-entity (entity)
  ((address :type address :accessor locatable-entity-address
            :display :full)
   (phone-number :type phone-number :display :terse)))



;;;; file time-zone.lisp
;;;; Define the type TIME-ZONE

(in-package "PLANNER")

;;; Integral number of hours west of Greenwich (Universal Time).
(deftype time-zone () `(integer 0 24))

(defvar *time-zone-table*
  '((eastern . 5)
    (central . 6)
    (mountain . 7)
    (pacific . 8)))

(defun find-time-zone (time-zone-name)
  (if (typep time-zone-name 'time-zone)
      time-zone-name
      (cdr (assoc time-zone-name *time-zone-table*
                  :test #'string-equal))))

(defun time-zone-name (time-zone)
  (or (car (find time-zone *time-zone-table* :key #'cdr))
      ;; Just return the number if there is no name.
      time-zone))
```

```
;;;; file state-object.lisp
;;;; Define the class STATE for Personal Planner

(in-package "PLANNER")

;;; A STATE is an entity representing a state in the United States,
;;; a Canadian province, or a foreign country.

(def-concrete-planner-class state (entity)
  ;; ABBREVIATION is the standard Postal Service two-letter
  ;; abbreviation.
  ((abbreviation :type string :accessor state-abbreviation)
   ;; TIME-ZONE is the predominant time zone in the state.
   ;; When a city is created, this time zone is used unless one
   ;; is supplied.
   (time-zone :type time-zone :accessor time-zone)
   ))

(defun make-state (name abbreviation time-zone)
  (let ((state (intern-instance 'state :name name)))
    (reintern-instance state :name abbreviation)
    (setf (state-abbreviation state) abbreviation)
    (setf (time-zone state) (if (typep time-zone 'time-zone)
                                time-zone
                                (find-time-zone time-zone)))
    state))

(defun find-state (name)
  (find-instance 'state :name name))



;;;; file address-object.lisp
;;;; Define the class ADDRESS for Personal Planner

(in-package "PLANNER")

;;; Addresses are usually found in the ADDRESS slot of
;;; LOCATABLE-ENTITIES.

(def-concrete-planner-class address (basic-planner-object)
  ((street-or-po-box :type string :display :terse)
   (city :type city :display :terse)
   (zip-code :type (integer 00000 99999) :display :terse)))
```

```
(defmethod print-object ((address address) stream)
  (with-slots (street-or-po-box city zip-code)
    address
    (format stream
            (if *print-escape*
                "#<Address ~A, ~S ~5,'0D>"
                "~A, ~A ~5,'0D")
            street-or-po-box
            city
            zip-code)))
```

```
;;;; file phone-number-object.lisp
;;;; Define the class PHONE-NUMBER for Personal Planner

(in-package "PLANNER")

(def-concrete-planner-class phone-number (basic-planner-object)
  ((area-code :type (integer 000 999))
   (exchange :type (integer 000 999))
   (local-number :type (integer 0000 9999))
   (extension :type (optional string))
   (type :type (member voice fax modem))))

(defmethod print-object ((object phone-number) stream)
  (with-slots (area-code exchange local-number extension type)
    object
    (format stream
            (if *print-escape*
                ;; Print phone number in this form:
                ;; (415) 555-8400.
                ;; If there's an extension, it becomes
                ;; (415) 555-8400 x8400.
                ;; If the type is FAX or MODEM, follow the
                ;; number with either (FAX) or (NUMBER).
                "#<Phone (~3,'0D)~3,'0D-~4,'0D~@[ x~A~]~@[ (~A)~]>"
                "(~3,'0D)~3,'0D-~4,'0D~@[ x~A~]~@[ (~A)~]")
            area-code exchange local-number extension
            (if (eq type 'voice) nil type))))
```

```
;;;; file city-object.lisp
;;;; Define the class CITY for Personal Planner

(in-package "PLANNER")

(def-concrete-planner-class city (entity)
  ((state :type state :accessor city-state :initarg :state)
   ;; Unless it is specified when the object is created, the time zone
   ;; is inherited from the state.
   (time-zone :type time-zone
              :accessor time-zone
              :initarg :time-zone)))

;;; This takes care of initializing the time-zone slot.  If it
;;; wasn't supplied by MAKE-INSTANCE, it gets defaulted from
;;; the state's time-zone.  It also takes a state name and turns
;;; it into a state object.
(defmethod shared-initialize :after ((city city) slot-names
                                     &rest initargs)
  (declare (ignore slot-names initargs))
  (when (slot-boundp city 'state)
    (let ((state (city-state city)))
      (when (stringp state)
        (setf (city-state city) (find-state state))))
    (unless (slot-boundp city 'time-zone)
      (setf (time-zone city) (time-zone (city-state city))))))

(defmethod print-object ((city city) stream)
  (format stream
          (if *print-escape*
              "#<~:(~A~) ~S "
              "~*~A, ")
          (class-name (class-of city))
          (entity-name city))
  (print-object (city-state city) stream)
  (when *print-escape*
    (write-char #\> stream)))

;;; This allows Menlo Park, California, and Menlo Park,
;;; New Jersey, to coexist by making the table key include
;;; the state as well as the city name.
(defmethod initargs-intern-key ((prototype city) &key name state)
  (list (class-name (class-of prototype))
        name
        (state-abbreviation (if (stringp state)
                                (find-state state) state))))
```

```
;;;; file person-object.lisp
;;;; Define the class PERSON for Personal Planner

(in-package "PLANNER")

(def-concrete-planner-class person (locatable-entity)
  ((organization :type organization :display :terse)
   (title :type string :display :terse)
   ))
```

```
;;;; file organization-object.lisp
;;;; Define the class ORGANIZATION for Personal Planner

(in-package "PLANNER")

(def-concrete-planner-class organization (locatable-entity)
  ()
  )
```

```
;;;; file event-object.lisp
;;;; Define the class EVENT for Personal Planner

(in-package "PLANNER")

;;; An event is something that can be placed in time,
;;; either in the past or the future.
(def-concrete-planner-class event (locatable-entity)
  ((starting-time :type universal-time
                  :accessor event-starting-time
                  :initarg :starting-time
                  :display :full)
   ;; The duration is in seconds.
   (duration :type elapsed-time :initform 0 :display :full)
   ))
```

```
(defmethod print-object ((object event) stream)
  (with-slots (starting-time location duration)
    object
    (format stream
            (if *print-escape*
                "#<~:(~A~): ~A at "
                "~*~:(~A~) at ")
            (class-name (class-of object))
                (if (slot-boundp object 'name)
                    (entity-name object)
                    "Anonymous"))
        (write-date-and-time starting-time stream)
        (when (> duration 0)
          (write-string " for " stream)
          (write-time-interval duration stream nil))
        (when *print-escape*
          (write-char #\> stream))))

;;; Events are identified by a combination of name and time.
(defmethod initargs-intern-key ((prototype event)
                                &key name starting-time)
  (list (class-name (class-of prototype))
        name
        starting-time))
```

```
;;;; file meeting-object.lisp
;;;; Define the class MEETING for Personal Planner

(in-package "PLANNER")

(def-concrete-planner-class meeting (event)
  ((attendee :type (list-of person) :display :full)))
```

```
;;;; file date-constants.lisp
;;;; Some constants required for parsing and printing dates

(in-package "PLANNER")

(defconstant month-names
  '#(() (January Jan) (February Feb) (March Mar) (April Apr) (May)
     (June Jun) (July Jul) (August Aug) (September Sept Sep)
```

```
    (October Oct) (November Nov) (December Dec))
  "The months of the year, in CL Decoded Time order.")

(defun month-name (month-index &optional short-p)
  (let ((entry (svref month-names month-index)))
    (if short-p
        (first (last entry))
        (first entry))))

(defun month-index (month-name)
  (position month-name month-names :test #'member))

(defconstant day-names
  '#((Monday Mon) (Tuesday Tue) (Wednesday Wed)
     (Thursday Thu) (Friday Fri) (Saturday Sat)
     (Sunday Sun))
  "The days of the week, in CL Decoded Time order.")

(defun day-name (day-index &optional short-p)
  (let ((entry (svref day-names day-index)))
    (if short-p
        (first (last entry))
        (first entry))))

(defun day-index (day-name)
  (position day-name day-names :test #'member))

(defconstant seconds-per-minute 60)

(defconstant minutes-per-hour 60)

(defconstant seconds-per-hour
  (* seconds-per-minute minutes-per-hour))

(defconstant hours-per-day 24)

(defconstant minutes-per-day
  (* minutes-per-hour hours-per-day))

(defconstant seconds-per-day
  (* seconds-per-hour hours-per-day))
```

```
;;;; file date-write.lisp
;;;; Functions for printing times and dates for people to read

(in-package "PLANNER")

(defun write-time-interval (seconds stream &optional short-p)
  ;; Write seconds to stream in the form:
  ;; 32 days, 4 hours, 5 minutes, 23 seconds
  (multiple-value-bind (days seconds)
      (truncate seconds seconds-per-day)
    (multiple-value-bind (hours seconds)
        (truncate seconds seconds-per-hour)
      (multiple-value-bind (minutes seconds)
          (truncate seconds seconds-per-minute)
        (if short-p
            ;; If short-p is true, use this form:
            ;; 32 04:05:23
            (format stream "~@[~D ~]~2,'0D:~2,'0D:~2,'0D"
                    (if (> days 0) days nil) hours minutes seconds)
            (let ((comma-needed nil))
              (flet ((print-component (amount name)
                       (when (> amount 0)
                         (when comma-needed
                           (write-string ", " stream))
                         (format stream "~D ~A" amount name)
                         (unless (eql amount 1)
                           (write-char #\s stream))
                         (setq comma-needed t))))
                (print-component days "day")
                (print-component hours "hour")
                (print-component minutes "minute")
                (print-component seconds "second")
                (unless comma-needed
                  ;; Nothing else has been printed, so it's an
                  ;; empty interval.
                  (write-string "0 seconds" stream)))))))))
        seconds)

(defun write-date-and-time (universal-time
                            stream
                            &key exclude-day exclude-time timezone)
  (multiple-value-bind (sec min hrs date month year
                        day daylight-p time-zone)
```

```
      (if timezone
          (decode-universal-time universal-time timezone)
          (decode-universal-time universal-time))
    (format stream
      ;; Print the day, followed by one space, if EXCLUDE-DAY is
      ;; false.  Print the date as a decimal string, padded on
      ;; the left with spaces; the month as a three-character
      ;; string; and the year as a two-character string, padded
      ;; on the left with zeros.  If EXCLUDE-TIME is false, print
      ;; the hours, minutes, and seconds as two-digit, zero-padded
      ;; fields, separated with colons.  Finally, print any time
      ;; zone as a three-character uppercase string.
      "~:[~*~;~@(~A~) ~]~2D-~@(~A~)-~2,'0D~@[~* ~2,'0D:~2,'0D:~2,
       '0D~@[ ~A~]~]"
      (not exclude-day)
      (day-name day t)
      date
      (month-name month t)
      (mod year 100)
      (not exclude-time)
      hrs
      min
      sec
      (if (<= 5 time-zone 8)
          (svref (if daylight-p
                     '#("EDT" "CDT" "MDT" "PDT")
                     '#("EST" "CST" "MST" "PST"))
                 (- time-zone 5))
          nil))))
```

```
;;;; file date-parse.lisp
;;;; A function for reading dates entered by people

(in-package "PLANNER")

(defun parse-date (string time-p)
  "Read a date from the string.  If time-p is true, read a time
   as well.  Returns an integer in Universal Time format.
   The date and time must be in the form: 3 April 1988 11:29 pm.
   Three-letter abbreviations are allowed for the month.  If a
   year is given between 50 and 99, 1900 will be added to it.
   If a year is given between 0 and 49, 2000 will be added."
```

```
(multiple-value-bind (day-of-month string-position)
    (parse-integer string :junk-allowed t)
  (check-type day-of-month (integer 1 31))
  (multiple-value-bind (month-name string-position)
      (read-from-string string t nil :start string-position)
    (check-type month-name symbol)
    (multiple-value-bind (year string-position)
        (parse-integer string :start string-position :junk-allowed t)
      (check-type year (integer 0 2100))
      (let ((month-number (month-index month-name)))
        (unless (and (integerp month-number) (< 0 month-number 13))
          (error "~S is not known to be the name of a month"
                 month-name))
        (let ((adjusted-year (if (< year 100)
                                 (if (< year 50)
                                     (+ year 2000)
                                     (+ year 1900))
                                 year)))
          (multiple-value-bind (hour minute)
              (if (not time-p)
                  (values 0 0)
                  (multiple-value-bind (hour string-position)
                      (parse-integer string
                                     :start string-position
                                     :junk-allowed t)
                    (check-type hour (integer 0 24))
                    ;; Skip over :
                    (multiple-value-bind (minute string-position)
                        (parse-integer string
                                       :start (1+ string-position)
                                       :junk-allowed t)
                      (check-type minute (integer 0 60))
                      ;; Look for AM or PM.  If neither, use
                      ;; 24-hour clock.
                      (let ((am-or-pm (read-from-string
                                       string nil string
                                       :start string-position)))
                        (values (if (eq am-or-pm 'pm)
                                    (+ hour 12)
                                    hour)
                                minute)))))
            (encode-universal-time 0 minute hour
                                   day-of-month month-number
                                   adjusted-year)))))))
```

```
(defconstant time-interval-table
  `(("second" . ,1)
    ("minute" . ,seconds-per-minute)
    ("hour" . ,seconds-per-hour)
    ("day" . ,seconds-per-day)))

(defun parse-time-interval (string)
  "Read a time interval as produced by write-time-interval.
 The format is either (optional) number of days followed by
 hours:minutes:seconds or the 'English' format,
 3 days, 13 hours, 4 minutes, 10 seconds."
;; It always begins with a number.
(multiple-value-bind (first-number string-position)
    (parse-integer string :junk-allowed t)
  (when (null first-number)
    (error "~A is not a time interval" string))
  ;; If the next non-whitespace character is a colon,
  ;; this is short format.
  (let ((string-length (length string))
        (non-whitespace-position
         (position-if-not #'(lambda (x)
                              (member x '(#\Space #\Tab)))
                          string
                          :start string-position)))
    (when (null non-whitespace-position)
      (error "~A is not a time interval"))
    (let ((next-char (aref string non-whitespace-position)))
      (cond ((char= next-char #\:)
             ;; This is short format with no days,
             ;; first-number is hours.
             (multiple-value-bind (minutes string-position)
                 (parse-integer string
                                :start (1+ non-whitespace-position)
                                :junk-allowed t)
               (+ (* first-number seconds-per-hour)
                  (* minutes seconds-per-minute)
                  (parse-integer string
                                 :start (1+ string-position)
                                 :junk-allowed t))))
```

```
        ((digit-char-p next-char)
         ;; This is the short format; first-number is days.
         (multiple-value-bind (hours string-position)
             (parse-integer string
                            :start non-whitespace-position
                            :junk-allowed t)
           (multiple-value-bind (minutes string-position)
               (parse-integer string
                              :start (1+ string-position)
                              :junk-allowed t)
             (+ (* first-number seconds-per-day)
                (* hours seconds-per-hour)
                (* minutes seconds-per-minute)
                (parse-integer string
                               :start (1+ string-position)
                               :junk-allowed t)))))
        (t
         ;; This is the long format, a series of numbers
         ;; followed by words.  We look for words in the
         ;; time interval table, multiply by the number of
         ;; seconds in that interval, and add to the total.
         (loop named outer
               with time-total = 0
               with string-position = non-whitespace-position
               with number = first-number
               do
               (loop
                 for entry in time-interval-table
                 ;; The CAR of an entry in the table is
                 ;; the name of the interval.
                 as name = (car entry)
                 as name-length = (length name)
                 as end-name-string-position
                    = (+ string-position name-length)
                 when (string-equal
                        string name
                        :start1 string-position
                        :end1
                        (min string-length
                             end-name-string-position))
               do
               (progn
                 ;; The CDR of the entry is the number
                 ;; of seconds in that interval.
                 (incf time-total (* number (cdr entry)))
                 ;; Skip over the letter "s", a comma and any
```

```
                ;; whitespace and then look for another
                ;; number.  If there is none, we are done.
                (let ((next-position
                       (position-if-not
                         #'(lambda (x)
                             (member x '(#\, #\s #\S
                                         #\Space #\Tab)))
                         string
                         :start end-name-string-position)))
                  (unless next-position
                    (return-from outer time-total))
                  (multiple-value-bind
                    (next-number next-position)
                      (parse-integer string
                                     :start next-position
                                     :junk-allowed t)
                    (unless next-number
                      (return-from outer time-total))
                    (setq number next-number)
                    (setq string-position
                          (position-if-not
                            #'(lambda (x)
                                (member x '(#\Space #\Tab)))
                            string
                            :start next-position))
                          (return))))
                finally
                (error "~A is not a time interval"
                       string))))))))
```

```
;;;; file toplevel.lisp
;;;; Top level for terminal-based user interaction for Personal Planner

(in-package "PLANNER")

(defun personal-planner ()
  ;; All interaction with Personal Planner happens in the PLANNER package.
  (let ((*package* (find-package "PLANNER")))
    ;; This loads the previous version of the database.
    (restore-planner-database t)
    ;; This CATCH is just here for the QUIT command to THROW to.
    (catch 'personal-planner
      (format t "~&Welcome to Personal Planner~%")
```

```
        (loop
          ;; This catches aborts from the debugger.
          (with-simple-restart
              (abort "Abort to Personal Planner Top Level")
            (loop
              (evaluate-command (read-command))))
          (format t "~&Back to Personal Planner Top Level~2%")))))

(defvar *command-table* '())

(defvar *command-list* '())

(defstruct command-table-entry
  (names '() :type list)
  (documentation "" :type string)
  function)

(defun read-command ()
  (accept `(terse-member ,@*command-list*) "command" "commands"))

(defun evaluate-command (user-entry)
  (funcall
    (command-table-entry-function (find-command user-entry))))

(defun find-command (user-entry)
  (dolist (table-entry *command-table* nil)
    (when (member user-entry
                  (command-table-entry-names table-entry)
                  :test #'eq)
      (return table-entry))))

(defmacro define-command
    (command-names user-documentation &body body)
  (let ((command-names (if (listp command-names)
                           command-names
                           (list command-names))))
    `(add-command ',command-names
                  ,user-documentation
                  #'(lambda ()
                      (format t "~&~A~%" ,user-documentation)
                      ,@body))))
```

```
(defun add-command (command-names user-documentation function)
  ;; Remove any existing commands of the same name.
  (dolist (table-entry *command-table*)
    (dolist (existing-command
             (command-table-entry-names table-entry))
      (when (member existing-command command-names :test #'eq)
        (setf (command-table-entry-names table-entry)
              (delete existing-command
                      (command-table-entry-names table-entry)
                      :test #'eq))))
    ;; If we removed all the names for this command, remove
    ;; the command.
    (when (null (command-table-entry-names table-entry))
      (setq *command-table*
            (delete table-entry *command-table* :test #'eq))))
;; Add the new command to the table.
(setf *command-table*
      (nconc *command-table*
             (list (make-command-table-entry
                    :names command-names
                    :documentation user-documentation
                    :function function))))
;; Remove existing names from the names list.
(dolist (new-name command-names)
  (setf *command-list*
        (delete new-name *command-list* :test #'eq)))
  ;; Add the command names to the names list.
  (setf *command-list*
        (nconc *command-list* (copy-list command-names)))
  command-names)

(define-command (help h) "Print this help message."
  (dolist (entry *command-table*)
    (format t "~&~{~S~^, ~}~30T~A~%"
            (command-table-entry-names entry)
            (command-table-entry-documentation entry))))

(define-command (add a) "Add a new entry."
(let* ((instance (prompt-for-new-entry)))
  (when (y-or-n-p "~A has been created.  Do you want to edit it?"
                  instance)
    (edit instance))))
```

```
(define-command (find f) "Find entries."
  (let ((instance (prompt-for-existing-entry)))
    (display instance)))

(define-command (change c) "Change an entry."
  (let ((instance (prompt-for-existing-entry)))
    (edit instance)))

(define-command (delete d) "Delete an entry."
  (multiple-value-bind (instance type initargs)
      (prompt-for-existing-entry)
    (declare (ignore type))
    (apply #'unintern-instance instance initargs)))

(define-command (rename r) "Rename an entry."
  (multiple-value-bind (instance type initargs)
      (prompt-for-existing-entry)
    (declare (ignore type))
    (apply #'unintern-instance instance initargs)
    (let ((initargs-copy (copy-list initargs)))
      (setf (getf initargs-copy :name) (accept 'string "new name"))
      (apply #'reintern-instance instance initargs-copy))))

(define-command (quit q)
  "Quit Personal Planner, saving the database."
  (save-planner-database t)
  (throw 'personal-planner t))

(define-command (nosave n) "Quit without saving the database."
  (throw 'personal-planner t))

(define-command (lisp l)
  "Evaluate a form and print the return values."
  (format t "~{~&~S~}"
          (multiple-value-list (eval (accept 't "expression")))))

(defun prompt-for-entry-type ()
  (accept `(terse-member ,@(concrete-planner-classes))
          "entry type"))

(defun prompt-for-existing-entry ()
  (accept-entity (prompt-for-entry-type) nil nil :existing))

(defun prompt-for-new-entry ()
  (accept-entity (prompt-for-entry-type) nil nil :new))
```

```
;;;; file type-dispatch.lisp
;;;; Define type dispatching functions

(in-package "PLANNER")

;;; These functions are used by DISPLAY, ACCEPT, and EDIT to maintain
;;; their database of type-specific operations.
(defun add-type-specific (type-name data database)
  (let* ((existing-entry (assoc type-name database :test #'equal))
         (database
          (if existing-entry
              (delete existing-entry database :test #'eq)
              database)))
    (cons (cons type-name data) database)))

(defun find-type-specific (type database)
  (loop for entry in database
        as entry-type = (car entry)
        with best-entry = nil
        and best-type = 't          ;The top of the type hierarchy
        do
        ;; An exact match is always taken.
        (when (or (equal type entry-type)
                  ;; If the CAR of type matches entry-type,
                  ;; e.g., for MEMBER or INTEGER, it's also okay.
                  (and (consp type)
                       (eq (car type) entry-type)))
  (return (cdr entry)))
;; Otherwise, check for subtypep.
(when (and (type-specifier-p entry-type)
           (subtypep type entry-type))
  ;; If it is, check if it is more specific than best-type.
  (when (subtypep entry-type best-type)
    (setq best-type entry-type)
    (setq best-entry (cdr entry))))
finally (return best-entry)))
```

```
;;;; file accept.lisp
;;;; Generalized user input

(in-package "PLANNER")

;;; ACCEPT is sort of a context-sensitive version of READ.  While
;;; READ can return values of any type, the type of input desired
```

```
;;; is specified in the call to ACCEPT.  Many types have
;;; specialized parsing in ACCEPT.  For example, (ACCEPT 'STRING)
;;; uses READ-LINE instead of READ.  Similarly, (ACCEPT 'PATHNAME)
;;; reads a line of input and turns it into a pathname.
(defun accept (type &optional singular-string plural-string)
  ;; TYPE is any legal type specifier.  A number of new types
  ;; have been defined, equivalent to existing types, just to
  ;; make this interface more useful.
  ;; SINGULAR-STRING describes the object to be entered, e.g.,
  ;; "file to compile," "address of this organization"
  ;; (where the referent of "this" is obvious from previous output).
  ;; PLURAL-STRING is a string that described a group of objects
  ;; of this type, e.g., "commands."
  ;; PLURAL-STRING is only used for MEMBER types, LIST-OF and
  ;; VECTOR. Note that in LIST-OF and VECTOR cases, SINGULAR-STRING
  ;; refers to an element of the aggregate to be entered.
  (let ((acceptor (find-specialized-acceptor type)))
    (if acceptor
        (funcall acceptor type singular-string plural-string)
        (default-acceptor type singular-string plural-string))))

;;; If the singular string prompt is not specified, we use the
;;; generic "object of type FOO."
(defun default-singular-string (singular-string type)
  (or singular-string (format nil "object of type ~S" type)))

;;; Similarly for the plural prompt.
(defun default-plural-string (plural-string type)
  (or plural-string (format nil "objects of type ~S" type)))

(defvar *specialized-acceptors* '())

(defun find-specialized-acceptor (type)
  ;; Find a type-specific acceptor function in the table.
  ;; *SPECIALIZED-ACCEPTORS*.
  (find-type-specific type *specialized-acceptors*))

(defmacro defacceptor (type-name args &body body)
  ;; Define a specialized acceptor for a particular type.
  ;; The function DEFINE-ACCEPTOR does all the work.
  `(define-acceptor ',type-name #'(lambda ,args ,@body)))

(defun define-acceptor (type-name function)
  (setq *specialized-acceptors*
        (add-type-specific type-name function *specialized-acceptors*))
  type-name)
```

```
(defun default-acceptor (type singular-string plural-string)
  ;; Used only if no specialized acceptor is defined.
  ;; Just call READ to get an object of the correct type.
  (declare (ignore plural-string))
  (prompt-accept type (default-singular-string singular-string type)
                 #'read))

;;; There is a bug in this scheme.  The type UNIVERSAL-TIME is
;;; equivalent to INTEGER.  However, UNIVERSAL-TIME has a
;;; specialized ACCEPT function.  If we just want to read an
;;; index, as in the LIST-OF acceptor, we might accidentally
;;; get the UNIVERSAL-TIME acceptor, which would be entirely
;;; wrong.  Therefore, we define an acceptor for INDEX that
;;; just does the default.
(define-acceptor 'index #'default-acceptor)

;;; PROMPT-ACCEPT is an auxiliary function that takes care of
;;; prompting the user, calling the reading function supplied by
;;; the caller, and checking that the result matches the type.
(defun prompt-accept (type prompt-string reader)
  (loop
    ;; Capitalize the first letter of the prompt.
    (format t "~@(~A~): " prompt-string)
    (force-output)
    (let ((result (funcall reader)))
      (when (typep result type)
        (return result))
      (format t "~&Sorry, ~S is not ~:[a~;an~] ~A~%"
              result (starting-vowel-p prompt-string)
              prompt-string))))

(defun starting-vowel-p (string)
  (member (char string 0) '(#\a #\e #\i #\o #\u)
          :test #'char-equal))

;;; Strings are accepted with READ-LINE.
(defacceptor string (type singular-string plural-string)
  (declare (ignore plural-string))
  (prompt-accept type (or singular-string "string") #'read-line))
```

```
;;; Pathnames are accepted with MERGE-PATHNAMES of READ-LINE.
(defacceptor pathname (type singular-string plural-string)
  (declare (ignore plural-string))
  (prompt-accept type (or singular-string "filename")
                 #'(lambda ()
                     (let ((string (read-line)))
                       (merge-pathnames string)))))

;;; MENU-MEMBER shows all the available choices and lets the user
;;; select one.  TERSE-MEMBER doesn't show all the choices, but
;;; defines ? to show the choices.  Plain MEMBER uses
;;; MENU-MEMBER if there are fewer than MENU-MEMBER-LIMIT
;;; elements; otherwise it uses TERSE-MEMBER.

;;; A very arbitrary limit.  Modify to taste.
(defparameter menu-member-limit 20)

(defun menu-member-acceptor (type singular-string plural-string)
  (declare (ignore singular-string))
  (format t "~&Select one of the following")
  (when plural-string
    (write-char #\Space)
    (write-string plural-string))
  (format t ":~%")
  (let* ((choice-list (rest type))
         (number-of-choices (length choice-list)))
    (loop for i from 1
          for choice in choice-list
          do (format t "~&~D: ~S~%" i choice))
    (let ((choice (accept `(or (integer 1 ,number-of-choices)
                               (member ,@choice-list))
                          "number or name" "numbers or names")))
      (if (integerp choice)
          (nth (1- choice) choice-list)
          choice))))

(define-acceptor 'menu-member #'menu-member-acceptor)

(defun terse-member-acceptor (type singular-string plural-string)
  (declare (ignore plural-string))
  (let ((choice-list (rest type))
        (prompt-string (or singular-string "option")))
```

```
    (loop
      (format t "~&~@(~A~) (? for options): " prompt-string)
      (force-output)
      (let ((result (read)))
        (cond ((eq result '?)
               (show-choices choice-list))
              ((member result choice-list)
               (return result))
              (t (format t
                         "Sorry, ~S is not ~:[a~;an~] ~A.
                          Type ? to see the available options"
                         result
                         (starting-vowel-p prompt-string)
                         prompt-string))))))

;;; We use the justification feature of FORMAT to make the list of
;;; choices more attractive.
(defun show-choices (choice-list)
  (format t "~&~{~<~%~1:;~S~>~^, ~}.~%" choice-list))

(define-acceptor 'terse-member #'terse-member-acceptor)

(defacceptor member (type singular-string plural-string)
  (if (< (length (rest type)) menu-member-limit)
      (menu-member-acceptor type singular-string plural-string)
      (terse-member-acceptor type singular-string plural-string)))

;;; (LIST-OF x) is just like LIST as a type specifier, but it
;;; allows us to make a list with specific types of elements.
(defun list-of-acceptor (type singular-string plural-string)
  ;; We call ACCEPT recursively for both the number of elements and
  ;; each individual element.
  (let* ((element-type (second type))
         (singular-string
          (default-singular-string singular-string element-type))
         (plural-string
          (default-plural-string plural-string element-type))
         (number-of-elements
          (accept 'index
                  (format nil "number of ~A" plural-string))))
    (loop for i from 1 to number-of-elements
          as indexed-singular-string = (format nil "~:R ~A"
                                               i singular-string)
          as indexed-plural-string = (format nil "~:R ~A"
                                             i plural-string)
          collect
```

```
            (accept element-type
                    indexed-singular-string
                    indexed-plural-string))))

(define-acceptor 'list-of #'list-of-acceptor)

;;; Vectors are read as lists and then converted at the end.
(defacceptor vector (type singular-string plural-string)
  (let* ((element-type (if (consp type) (second type) 't))
         (list-result (list-of-acceptor `(list-of ,element-type)
                                        singular-string
                                        plural-string)))
    (coerce list-result type)))

;;; (OPTIONAL type) means (OR type NULL).  First ask if this
;;; is going to be supplied, and then recursively ACCEPT.
(defacceptor optional (type singular-string plural-string)
  (let ((optional-type (second type)))
    (and (y-or-n-p "Is there ~:[a~;an~] ~A?"
                   (starting-vowel-p singular-string)
                   (default-singular-string singular-string
                                            optional-type))
         (accept optional-type singular-string plural-string))))

;;; BASIC-PLANNER-OBJECTs have slot type information saved away.
;;; This is used to prompt for initargs to create them.
(defacceptor basic-planner-object
    (type singular-string plural-string)
  (declare (ignore plural-string))
  (let ((type-string (pretty-symbol-name type)))
    (format t "~&~@(~A~):~%" (or singular-string type-string))
    (let ((object (make-instance type)))
      (initialize-planner-slots object type type-string))))

(defun slot-singular-string (slot-name type-string)
  (format nil "~A for this ~A" (pretty-symbol-name slot-name)
          type-string))

(defun slot-plural-string (slot-name type-string)
  (format nil "~As for this ~A" (pretty-symbol-name slot-name)
          type-string))

(defun pretty-symbol-name (symbol)
  ;; Lowercase all characters, and turn hyphens into spaces.
  (substitute #\Space #\- (string-downcase (symbol-name symbol))))
```

```
(defun initialize-planner-slots (object type type-string)
  (let* ((slot-information (get-slot-information type))
         (slot-values
          (accept-slot-values slot-information type-string)))
    (loop for value in slot-values
          for slot-info in slot-information
          as slot-name = (slot-description-name slot-info)
          do (setf (slot-value object slot-name) value))
    object))

(defun accept-slot-values (slot-information type-string)
  (loop for slot-info in slot-information
        collect (accept-slot-value slot-info type-string)))

(defun accept-slot-value (slot-info type-string)
  (let* ((slot-name (slot-description-name slot-info))
         (slot-type (slot-description-type slot-info))
         (singular-string
          (slot-singular-string slot-name type-string))
         (plural-string
          (slot-plural-string slot-name type-string)))
    (accept slot-type singular-string plural-string)))

(defun accept-initargs (slot-information type-string)
  ;; Returns a list (:INITARG value :INITARG value) in a form
  ;; suitable to use with APPLY.
  (loop for slot-info in slot-information
        as initarg = (slot-description-initarg slot-info)
        when initarg
        collect initarg
        and collect (accept-slot-value slot-info type-string)))

(defun accept-entity (type singular-string plural-string
                           &optional existing-or-new)
  (declare (ignore plural-string))
  ;; If EXISTING-OR-NEW is NIL, either one is acceptable. If it is
  ;; :NEW, only new entities will be accepted.  If it is :EXISTING,
  ;; only existing entities will be accepted.
  ;; Three values are returned: the entity, its type, and the
  ;; initargs.
  (let ((type-string (pretty-symbol-name type)))
    (format t "~&~@(~@[~A ~]~A~):~%" existing-or-new
            (or singular-string type-string))
```

```
        (let* ((slot-information (get-slot-information type))
               (initargs
                (accept-initargs slot-information type-string))
               (existing-instance
                (apply #'find-instance type initargs))
               (instance
                (if existing-instance
                    (if (eq existing-or-new :new)
                        (error "That entry already exists")
                        existing-instance)
                    (if (eq existing-or-new :existing)
                        (error "That entry doesn't exist")
                        (apply #'intern-instance type initargs)))))
          (values instance type initargs))))

(define-acceptor 'entity #'accept-entity)

;;; A time-zone is either a defined name or the number of
;;; hours west of GMT.
(defacceptor time-zone (type singular-string plural-string)
  (declare (ignore plural-string))
  (prompt-accept type (or singular-string "time zone")
                 #'(lambda ()
                     (find-time-zone (read)))))

;;; Universal time is parsed using PARSE-DATE.
(defacceptor universal-time (type singular-string plural-string)
  (declare (ignore plural-string))
  (prompt-accept type (or singular-string "time")
                 #'(lambda ()
                     (let ((string (read-line)))
                       (parse-date string t)))))

;;; Elapsed time is parsed using PARSE-TIME-INTERVAL.
(defacceptor elapsed-time (type singular-string plural-string)
  (declare (ignore plural-string))
  (prompt-accept type (or singular-string "time interval")
                 #'(lambda ()
                     (let ((string (read-line)))
                       (parse-time-interval string)))))
```

```
;;;; file edit.lisp
;;;; Modify existing objects

(in-package "PLANNER")

;;; (EDIT object type) takes an existing object and returns an
;;; object (possibly the same object) that has been modified
;;; by the user. If there is no edit method, the user is
;;; prompted for a new value using ACCEPT.
(defun edit (object &optional type singular-string plural-string)
  (let* ((type (or type (type-of object)))
         (editor (find-editor object type)))
    (if editor
        (funcall editor object type singular-string plural-string)
        (accept type singular-string plural-string))))

(defvar *editors* '())

(defun find-editor (object type)
  (declare (ignore object))
  (find-type-specific type *editors*))

(defmacro defeditor (type-name args &body body)
  `(define-editor ',type-name
      #'(lambda ,args ,@body)))

(defun define-editor (type-name function)
  (setq *editors* (add-type-specific type-name function *editors*))
  type-name)

(defeditor basic-planner-object
    (object type singular-string plural-string)
  (declare (ignore singular-string plural-string))
  (let* ((slot-info (get-slot-information type))
         (slot-names (slot-information-names slot-info)))
    (loop for slot-to-change =
          (accept `(terse-member ,@slot-names done)
                  "slot to change"
                  "slots to change")
    until (eq slot-to-change 'done)
    do (edit-slot object slot-to-change
                  (find-slot-description slot-to-change
                                         slot-info)))))
```

```
(defun edit-slot (object slot-to-change slot-info)
  (let* ((slot-type (slot-description-type slot-info))
         (replace-or-modify
          (if (slot-boundp object slot-to-change)
              (progn (format t "~&Current value is:~%")
                     (display (slot-value object slot-to-change)
                              slot-type)
                     (accept '(terse-member replace modify)
                             "replace or modify this value"))
              'replace)))
    (setf (slot-value object slot-to-change)
          (if (eq replace-or-modify 'replace)
              (accept slot-type slot-to-change)
              (edit (slot-value object slot-to-change)
                    slot-type
                    (slot-singular-string slot-to-change
                                          slot-type))))))
```

```
;;;; file display.lisp
;;;; Data-driven display function

(in-package "PLANNER")

;;; DISPLAY is the output equivalent of ACCEPT, i.e., PRINT is to
;;; READ as DISPLAY is to ACCEPT.  A type can be specified to
;;; DISPLAY to use a type-specific display function.
(defun display (object &optional type)
  (let* ((type (or type (type-of object)))
         (displayer (find-displayer type)))
    (if displayer
        (funcall displayer object type)
        ;; Default display is just PRINT.
        (format t "~&~A~%" object))))

(defvar *displayers* '())

(defun find-displayer (type)
  (find-type-specific type *displayers*))

(defmacro defdisplay (type-name args &body body)
  `(define-display ',type-name
       #'(lambda ,args ,@body)))
```

```
(defun define-display (type-name function)
  (setq *displayers*
        (add-type-specific type-name function *displayers*))
  type-name)

;;; This is the default display function for BASIC-PLANNER-OBJECT,
;;; that is, for those classes that do not have more specific
;;; display functions.  It just prints the slots in order.
(defdisplay basic-planner-object (object type)
  (display-slots object type))

(defun display-slots (object type)
  (let ((slot-info (get-slot-information type)))
    (dolist (slot slot-info)
      (let ((display-info (slot-description-display slot))
            (slot-name (slot-description-name slot))
            (slot-type (slot-description-type slot)))
        ;; If there's no display info or nothing in slot, print nothing.
        (when (and display-info (slot-boundp object slot-name))
          (case display-info
            (:full
             (display (slot-value object slot-name) slot-type))
            (:terse
             (format t "~&~A~%" (slot-value object slot-name)))))))))

(defdisplay list-of (object type)
  (let* ((element-type (second type)) (elements (length object)))
    (format t "~&~D ~A~2:*~P~2%"
              elements (pretty-symbol-name element-type))
    (dolist (element object)
      (display element element-type)
      (terpri))))

;;; Universal time is displayed using WRITE-DATE-AND-TIME.
(defdisplay universal-time (object type)
  (declare (ignore type))
  (fresh-line)
  (write-date-and-time object t)
  (terpri))

;;; Elapsed time is displayed using WRITE-TIME-INTERVAL.
(defdisplay elapsed-time (object type)
  (declare (ignore type))
  (fresh-line)
  (write-time-interval object t)
  (terpri))
```

```
;;;; file database.lisp
;;;; Interface for saving and restoring the Personal Planner database

(in-package "PLANNER")

(defvar *database-file* "planner.data")

(defun restore-planner-database (verbose)
  (when verbose
    (format t "~&Loading database..."))
  (setf (slot-value
          (class-prototype (find-class 'entity)) 'intern-table)
        (read-object-from-file *database-file*))
  (when verbose
    (format t "done~%")))

(defun save-planner-database (verbose)
  (when verbose
    (format t "~&Saving database..."))
  (write-object-to-file
    (slot-value
      (class-prototype (find-class 'entity)) 'intern-table)
    *database-file*)
  (when verbose
    (format t "done~%")))


;;;; file file-constants.lisp
;;;; Constants needed for file saving and restoring

(in-package "PLANNER")

;;; A "planner save file" is a description of a single Lisp object
;;; and all objects contained within it.  It is similar to the
;;; printed representation of the object in that it descends
;;; structure down to symbols, numbers, and characters.  It preserves
;;; structure sharing and circularity within the object.  It should
;;; be quite a bit faster than printing with *PRINT-CIRCLE* set to T,
;;; and it can save the structure of some objects that PRINT doesn't
;;; handle, such as hash tables.

;;; It is a character file consisting of a sequence of Lisp objects
;;; parsed by READ.

(defconstant cons-constructor-code 1)
```

```
(defconstant vector-constructor-code 2)

(defconstant hash-table-constructor-code 3)

(defconstant standard-object-constructor-code 4)

(defconstant simple-object-code 5)

(defconstant table-entry-code 6)

(defconstant table-reference-code 7)

(defconstant single-reference-code 8)



;;;; file file-save.lisp
;;;; Saves a data structure into a file

(in-package "PLANNER")

;;; A SAVE-TABLE keeps track of the shared structure within
;;; the data structure being saved.
(defstruct save-table
  ;; HASH-TABLE contains an entry for each element in the data
  ;; structure. The entry is T when the object is first
  ;; encountered and is set to the current OBJECT-COUNT when
  ;; it is encountered a second time.
  (hash-table (make-hash-table))
  ;; OBJECT-COUNT is a count of all elements encountered more
  ;; than once.
  (object-count 0))

;;; This function provides a functional interface to the
;;; hash table in the save table.
(defun get-table-entry (object table)
  (gethash object (save-table-hash-table table)))

(defun set-table-entry (object table value)
  (setf (gethash object (save-table-hash-table table)) value))

(defsetf get-table-entry set-table-entry)

;;; WRITE-OBJECT-TO-FILE is the program interface to data
;;; structure saving.  It takes an object and a file name
```

```
;;; and saves the object in the file such that it can be
;;; restored using READ-OBJECT-FROM-FILE.
(defun write-object-to-file (object file)
  (let ((table (make-save-table)))
    (find-multiple-references object table)
    (save-object object file table)))

;;; Put this node of the data structure into the save table,
;;; allocating object numbers to those that are encountered
;;; multiple times.
(defun find-multiple-references (object table)
  (let ((entry (get-table-entry object table)))
    (case entry
      ((nil)
       (setf (get-table-entry object table) t)
       (find-recursive-references object table))
      ((t)
       (setf (get-table-entry object table)
             (save-table-object-count table))
       (incf (save-table-object-count table))))))

;;; Actually descend each data structure, calling
;;; FIND-MULITPLE-REFERENCES on each element.
(defun find-recursive-references (object table)
  (typecase object
    (cons
     ;; For a cons cell, we check the car and the cdr.
     (find-multiple-references (car object) table)
     (find-multiple-references (cdr object) table))
    ((or character number string symbol)
     ;; Strings, characters, numbers and symbols are not
     ;; considered to have components.
     )
    (hash-table
     ;; For a hash table, we check the hash table's test and every
     ;; key and value stored in the table.
     (find-multiple-references (hash-table-test object) table)
     (maphash
       #'(lambda (key value)
           (find-multiple-references key table)
           (find-multiple-references value table))
       object))
```

```
    (standard-object
     ;; For standard-objects (CLOS instances), we check the class
     ;; of the instance and every slot name and slot that is
     ;; not unbound.
     (let* ((class (class-of object))
            (name (class-name class))
            (named-class (and name (find-class name))))
       ;; If named-class and class are the same, use the name
       ;; instead of the class object.
       (find-multiple-references (if named-class name class) table)
       (let* ((slots (class-slots class)))
         (dolist (slot-description slots)
           (let ((name (slotd-name slot-description)))
             (when (slot-boundp object name)
               (find-multiple-references name table)
               (find-multiple-references
                 (slot-value object name) table))))))
    (vector
     ;; Each element of a vector is checked.
     (let ((len (length object)))
       (dotimes (i len)
         (find-multiple-references (aref object i) table))))
    (t
         ;; There are many other objects that might be encountered,
         ;; such as structures, but we have no way to save them.  It is
         ;; possible to add special case code for particular data types.
         (error "There is no way to save a ~S in a file" object))))

;;; Actually write out the data after doing the tree walk.
(defun save-object (object file table)
  (with-open-file (stream file :direction :output)
    (write-table-size (save-table-object-count table) stream)
    (save-object-to-stream object stream table)))

(defun save-object-to-stream (object stream table)
  (let ((entry (get-table-entry object table)))
    (cond ((already-saved-p entry)
           (write-backward-reference entry stream))
          (t
           (cond ((multiple-reference-p entry)
                  (write-enter-multiple-reference entry stream)
                  (mark-as-already-saved object table entry))
                 (t (write-enter-single-reference stream)))
           (save-actual-object object stream table)))))
```

```
(defun save-actual-object (object stream table)
  (typecase object
    (cons
     (write-constructor-code cons-constructor-code stream)
     (save-object-to-stream (car object) stream table)
     (save-object-to-stream (cdr object) stream table))
    ((or character number string symbol)
     (write-simple-object object stream))
    (hash-table
     (write-constructor-code hash-table-constructor-code stream)
     (save-object-to-stream (hash-table-test object) stream table)
     (write-length (hash-table-count object) stream)
     (maphash
       #'(lambda (key value)
           (save-object-to-stream key stream table)
           (save-object-to-stream value stream table))
       object))
    (standard-object
     (write-constructor-code standard-object-constructor-code
                             stream)
     (let* ((class (class-of object))
            (name (class-name class))
            (named-class (and name (find-class name))))
       ;; If named-class and class are the same, use the name
       ;; instead of the class object.
       (save-object-to-stream (if (eq class named-class)
                                  name
                                  class)
                              stream table)
       (let* ((slots (class-slots class))
              (slot-count (loop for slotd in slots
                                as name = (slotd-name slotd)
                                count (slot-boundp object name))))
         (write-length slot-count stream)
         (dolist (slot-description slots)
           (let ((name (slotd-name slot-description)))
             (when (slot-boundp object name)
               (save-object-to-stream name stream table)
               (save-object-to-stream (slot-value object name)
                                      stream table)))))))
    ((vector t)
     (write-constructor-code vector-constructor-code stream)
     (let ((len (length object)))
       (write-length len stream)
       (dotimes (i len)
         (save-object-to-stream (aref object i) stream table))))))
```

```
(defun already-saved-p (entry)
  (and (integerp entry)
       (< entry 0)))

(defun mark-as-already-saved (object table entry)
  (setf (get-table-entry object table) (- (1+ entry))))

(defun write-backward-reference (entry stream)
  (write-one-element table-reference-code stream)
  (write-one-element (1- (- entry)) stream))

(defun multiple-reference-p (entry)
  (not (eql entry t)))

(defun write-enter-multiple-reference (entry stream)
  (write-one-element table-entry-code stream)
  (write-one-element entry stream))

(defun write-enter-single-reference (stream)
  (write-one-element single-reference-code stream))

(defun write-table-size (count stream)
  (write-one-element count stream))

(defun write-constructor-code (code stream)
  (write-one-element code stream))

(defun write-length (length stream)
  (write-one-element length stream))

(defun write-simple-object (object stream)
  (write-one-element simple-object-code stream)
  (write-one-element object stream))

(defun write-one-element (object stream)
  (prin1 object stream)
  (terpri stream))
```

```
;;;; file file-restore.lisp
;;;; Restore a data structure from a file

(in-package "PLANNER")

(defun read-object-from-file (file)
  (with-open-file (stream file)
    (read-object-from-stream stream)))

(defun read-object-from-stream (stream)
  (let* ((table-size (read-one-element stream))
         (table (make-array table-size)))
    (construct-object-from-stream table stream)))

(defun construct-object-from-stream (table stream)
  (let ((entry-code (read-one-element stream)))
    (select entry-code
      (table-entry-code
        (read-object-contents table stream
                              (read-one-element stream)))
      (single-reference-code
        (read-object-contents table stream nil))
      (table-reference-code
        (svref table (read-one-element stream))))))

(defun read-object-contents (table stream table-index)
  (let ((constructor-code (read-one-element stream)))
    (select constructor-code
      (cons-constructor-code
        (let ((object (cons nil nil)))
          (when table-index
            (setf (svref table table-index) object))
          (setf (car object)
                (construct-object-from-stream table stream))
          (setf (cdr object)
                (construct-object-from-stream table stream))
          object))
      (hash-table-constructor-code
        (let ((object (make-hash-table
                       :test
                       (construct-object-from-stream table stream)))
              (count (read-length stream)))
```

```
        (dotimes (i count)
          (setf (gethash (construct-object-from-stream table
                                                       stream)
                         object)
                (construct-object-from-stream table stream)))
        object))
    (standard-object-constructor-code
      (let* ((class (construct-object-from-stream table stream))
             (object (make-instance class))
             (slot-count (read-length stream)))
        (when table-index
          (setf (svref table table-index) object))
        (dotimes (i slot-count)
          (setf (slot-value object
                            (construct-object-from-stream table
                                                          stream))
                (construct-object-from-stream table stream)))
        object))
    (vector-constructor-code
      (let* ((length (read-length stream))
             (vector (make-array length)))
        (when table-index
          (setf (svref table table-index) vector))
          (dotimes (i length)
            (setf (svref vector i)
                  (construct-object-from-stream table stream)))
          vector))
        (simple-object-code
          (let ((object (read-simple-object stream)))
            (when table-index
              (setf (svref table table-index) object))
            object)))))

(defun read-length (stream)
  (read stream))

(defun read-one-element (stream)
  (read stream))

(defun read-simple-object (stream)
  (read stream))
```

Appendix B

A SAMPLE SESSION

This appendix contains a sample session with the *Personal Planner*.

```
;; We begin by loading in the file load.lisp.
> (load "load.lisp")
;;; Loading source file load.lisp
#P"/lucid11/eb/book/lisp/load.lisp"

;; We specify that we want to compile and load the other program
;; files and the current database.
> (load-planner :compile)
;;; You are using the compiler in development mode
;;; (compilation-speed = 3)
;;; Generation of full safety checking code is enabled
;;; (safety = 3)
;;; Optimization of tail calls is disabled (speed = 2)
;;; Reading source file "package.lisp"
;;; Writing binary file "package.lbin"
;;; Loading binary file "package.lbin"
;;; Reading source file "plist.lisp"
;;; Writing binary file "plist.lbin"
;;; Loading binary file "plist.lbin"
;;; Reading source file "extra-types.lisp"
;;; Writing binary file "extra-types.lbin"
;;; Loading binary file "extra-types.lbin"
;;; Reading source file "planner-object.lisp"
;;; Writing binary file "planner-object.lbin"
;;; Loading binary file "planner-object.lbin"
;;; Reading source file "interned-mixin.lisp"
;;; Writing binary file "interned-mixin.lbin"
;;; Loading binary file "interned-mixin.lbin"
;;; Reading source file "entity-object.lisp"
```

```
;;; Writing binary file "entity-object.lbin"
;;; Loading binary file "entity-object.lbin"
;;; Reading source file "time-zone.lisp"
;;; Writing binary file "time-zone.lbin"
;;; Loading binary file "time-zone.lbin"
;;; Reading source file "state-object.lisp"
;;; Writing binary file "state-object.lbin"
;;; Loading binary file "state-object.lbin"
;;; Reading source file "address-object.lisp"
;;; Writing binary file "address-object.lbin"
;;; Loading binary file "address-object.lbin"
;;; Reading source file "phone-number-object.lisp"
;;; Writing binary file "phone-number-object.lbin"
;;; Loading binary file "phone-number-object.lbin"
;;; Reading source file "city-object.lisp"
;;; Writing binary file "city-object.lbin"
;;; Loading binary file "city-object.lbin"
;;; Reading source file "person-object.lisp"
;;; Writing binary file "person-object.lbin"
;;; Loading binary file "person-object.lbin"
;;; Reading source file "organization-object.lisp"
;;; Writing binary file "organization-object.lbin"
;;; Loading binary file "organization-object.lbin"
;;; Reading source file "event-object.lisp"
;;; Writing binary file "event-object.lbin"
;;; Loading binary file "event-object.lbin"
;;; Reading source file "meeting-object.lisp"
;;; Writing binary file "meeting-object.lbin"
;;; Loading binary file "meeting-object.lbin"
;;; Reading source file "date-constants.lisp"
;;; Writing binary file "date-constants.lbin"
;;; Loading binary file "date-constants.lbin"
;;; Reading source file "date-write.lisp"
;;; Writing binary file "date-write.lbin"
;;; Loading binary file "date-write.lbin"
;;; Reading source file "date-parse.lisp"
;;; Writing binary file "date-parse.lbin"
;;; Loading binary file "date-parse.lbin"
;;; Reading source file "toplevel.lisp"
;;; Writing binary file "toplevel.lbin"
;;; Loading binary file "toplevel.lbin"
;;; Reading source file "type-dispatch.lisp"
;;; Writing binary file "type-dispatch.lbin"
;;; Loading binary file "type-dispatch.lbin"
;;; Reading source file "accept.lisp"
;;; Writing binary file "accept.lbin"
```

```
;;; Loading binary file "accept.lbin"
;;; Reading source file "edit.lisp"
;;; Writing binary file "edit.lbin"
;;; Loading binary file "edit.lbin"
;;; Reading source file "display.lisp"
;;; Writing binary file "display.lbin"
;;; Loading binary file "display.lbin"
;;; Reading source file "database.lisp"
;;; Writing binary file "database.lbin"
;;; Loading binary file "database.lbin"
;;; Reading source file "file-constants.lisp"
;;; Writing binary file "file-constants.lbin"
;;; Loading binary file "file-constants.lbin"
;;; Reading source file "file-save.lisp"
;;; Writing binary file "file-save.lbin"
;;; Loading binary file "file-save.lbin"
;;; Reading source file "file-restore.lisp"
;;; Writing binary file "file-restore.lbin"
;;; Loading binary file "file-restore.lbin"
T

;; We make certain that we are in the correct package.
> (in-package 'planner)
#<Package "PLANNER" 79EC0B>

;; Now, we begin our interaction with the Personal Planner.
> (personal-planner)
Loading database...done
Welcome to Personal Planner
Command (? for options): ?
HELP, H, ADD, A, FIND, F, CHANGE, C, DELETE, D, RENAME, R, QUIT, Q,
NOSAVE, N, LISP, L.
Command (? for options): help
Print this help message.
HELP, H                 Print this help message.
ADD, A                  Add a new entry.
FIND, F                 Find entries.
CHANGE, C               Change an entry.
DELETE, D               Delete an entry.
RENAME, R               Rename an entry.
QUIT, Q                 Quit Personal Planner, saving the database.
NOSAVE, N               Quit without saving the database.
LISP, L                 Evaluate a form and print the return values.

;; We search for current entries in the database.
Command (? for options): find
```

```
Find entries.
Entry type (? for options): person
Existing person:
Name for this person: Eric Benson
Eric Benson
Lucid, Inc.
Principal Scientist

;; We decide to modify an entry.
Command (? for options): change
Change an entry.
Entry type (? for options): person
Existing person:
Name for this person: Eric Benson
Slot to change (? for options): ?
NAME, NICKNAMES, ADDRESS, PHONE-NUMBER, ORGANIZATION, TITLE, DONE.
Slot to change (? for options): phone-number
Phone-number:
Area code for this phone number: 415
Exchange for this phone number: 555
Local number for this phone number: 1389
Is there an extension for this phone number? (Y or N): y
Extension for this phone number: 5523
Select one of the following types for this phone number:
1: VOICE
2: FAX
3: MODEM
Number or name: 1
Slot to change (? for options): done

;; We try to add an entry that already exists.
Command (? for options): add
Add a new entry.
Entry type (? for options): person
New person:
Name for this person: Molly Miller
Molly Miller has been created.  Do you want to edit it? (Y or N): y
Slot to change (? for options): ?
NAME, NICKNAMES, ADDRESS, PHONE-NUMBER, ORGANIZATION, TITLE, DONE.
Slot to change (? for options): organization
Organization:
Name for this organization: Lucid, Inc.
Slot to change (? for options): title
Title: Manager of Technical Publications
Slot to change (? for options): phone-number
Phone-number:
```

```
Area code for this phone number: 415
Exchange for this phone number: 555
Local number for this phone number: 1389
Is there an extension for this phone number? (Y or N): y
Extension for this phone number: 5599
Select one of the following types for this phone number:
1: VOICE
2: FAX
3: MODEM
Number or name: voice
Slot to change (? for options): done

;; Now we look at the modified entry.
Command (? for options): find
Find entries.
Entry type (? for options): person
Existing person:
Name for this person: Molly Miller
Molly Miller
(415)555-1389 x5599
Lucid, Inc.
Manager of Technical Publications

;; We try to schedule a meeting.
Command (? for options): add
Add a new entry.
Entry type (? for options): meeting
New meeting:
Name for this meeting: Staff Meeting
Starting time for this meeting: 20 Sep 1989 2:00 pm
Staff Meeting at Wed 20-Sep-89 14:00:00 PDT has been created.
Do you want to edit it? (Y or N): y
Slot to change (? for options): ?
NAME, NICKNAMES, ADDRESS, PHONE-NUMBER, STARTING-TIME, DURATION,
ATTENDEE, DONE.
Slot to change (? for options): attendee
Number of objects of type person: 2
First attendee:
Name for this person: Eric Benson
Second attendee:
Name for this person: Molly Miller
Slot to change (? for options): duration
Current value is:
0 seconds
Replace or modify this value (? for options): replace
Duration: 1 hour
```

```
Slot to change (? for options): done

;; We examine the modified entry.
Command (? for options): find
Find entries.
Entry type (? for options): meeting
Existing meeting:
Name for this meeting: Staff Meeting
Starting time for this meeting: 20 Sep 1989 2:00 pm
Staff Meeting
Wed 20-Sep-89 14:00:00 PDT
1 hour
2 persons

Eric Benson
(415)555-1389 x5523
Lucid, Inc.
Principal Scientist

Molly Miller
(415)555-1389 x5599
Lucid, Inc.
Manager of Technical Publications

;; Now, we're done.
Command (? for options): quit
Quit Personal Planner, saving the database.
Saving database...done
T

;; Let's check that the database has stored the information
;; correctly by starting up again.
> (personal-planner)
Loading database...done
Welcome to Personal Planner
Command (? for options): find
Find entries.
Entry type (? for options): meeting
Existing meeting:
Name for this meeting: Staff Meeting
Starting time for this meeting: 20 Sep 1989 2:00 pm
Staff Meeting
Wed 20-Sep-89 14:00:00 PDT
1 hour
2 persons
```

```
Eric Benson
(415)555-1389 x5523
Lucid, Inc.
Principal Scientist

Molly Miller
(415)555-1389 x5599
Lucid, Inc.
Manager of Technical Publications

;; Looks like it worked.
Command (? for options): quit
Quit Personal Planner, saving the database.
Saving database...done
T
> (quit)
```

Appendix C

SUGGESTED EXERCISES

This appendix contains suggested exercises that are based on the *Personal Planner*.

1. Add more entity and event types.
2. Add recurring events, for weekly, monthly, quarterly, or annual events.
3. Add a facility for tracking expenses.
4. Revise the date-parsing mechanism so that it allows the user more flexibility in specifying dates.
5. Add an automatic reminder facility to remind the user about upcoming events.
6. Write a more English-like command interface.
7. Write a window-based user interface that provides menus of options, is mouse active, and provides an edit window for directly updating entries in the database.
8. Add commands for searching the database:
   a. Find all events in a certain time period.
   b. Search for entries matching some substring.
9. Improve the database storage mechanism:
   a. Use a binary file instead of a character file for database storage.
   b. Use a standard database interface that only loads into Lisp memory the objects used in one session rather than the entire database.
10. Revise all code parsing by first writing a "parser generator," such as the UNIX yacc facility.

BIBLIOGRAPHY

Abelson, Harold, and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. Cambridge, Mass.: MIT Press, 1985.

Anderson, John R., Albert T. Corbett, and Brian J. Reiser. *Essential Lisp*. Reading, Mass.: Addison-Wesley, 1987.

Barr, Avron, and Edward A. Feigenbaum, eds. *The Handbook of Artificial Intelligence*. Los Altos, Calif.: William Kaufmann, 1982.

Bobrow, Daniel G., and Gregor Kiczales. "Common Lisp Object Specification: Metaobject Protocol." X3J13 Draft Document 89-003. December 1988.

Brooks, Rodney A. *Programming in Common Lisp*. New York: Wiley, 1985.

"CLX—Common Lisp Language X Interface." Texas Instruments, Inc., Dallas, Texas, 1988.

Haskell, Jo A. L., and Molly M. Miller. *The Common Lisp Object System*. Bedford, Mass.: Digital Press, forthcoming Summer 1990.

Hearn, Anthony C. *REDUCE User's Manual*. The Rand Corporation, Santa Monica, California, 1987.

Keene, Sonya A. *Object-Oriented Programming in Common Lisp*. Reading, Mass.: Addison-Wesley, 1989.

Kernighan, Brian W., and P. J. Plauger. *The Elements of Programming Style*. New York: McGraw-Hill, 1974.

———. *Software Tools*. Reading, Mass.: Addison-Wesley, 1976.

Kessler, Robert R. *Lisp Objects and Symbolic Programming*. Glenview, Ill.: Scott, Foresman, 1988.

Ledgard, Henry. *Professional Software Programming Practice*. Reading, Mass.: Addison-Wesley, 1987.

Lucid Common Lisp Advanced User's Guide. Lucid, Inc., Menlo Park, California, 1988.

Moses, J. "A MACSYMA Primer." Math. Lab. Memo. No. 2. Computer Science Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1975.

Paepcke, Andreas. "PCLOS: A Critical Review." Programming Languages Department, Hewlett-Packard Laboratories, Palo Alto, California, March 17, 1989.

Steele, Guy L., Jr. *Common Lisp: The Language*. 2d ed. Bedford, Mass.: Digital Press, 1990.

Tatar, Deborah G. *A Programmer's Guide to Common Lisp*. Bedford, Mass.: Digital Press, 1987.

Winston, Patrick Henry, and Berthold Klaus Paul Horn. *Lisp*. 3d ed. Reading, Mass.: Addison-Wesley, 1989.

Zimmer, J. A. *Abstraction for Programmers*. New York: McGraw-Hill, 1985.

INDEX

' (quote), 60
` (backquote), 60
`, (backquote comma), 60
`,@ (backquote comma at-sign), 60
#' (sharp quote), 60
#| (sharp bar), 105
; (semicolon), 105
;; (two semicolons), 104–105
;;; (three semicolons), 102–103, 104
;;;; (four semicolons), 104
= (equal sign), 58
() (parentheses), 41

Abstraction, 65–88
  basics of, 66–67
  data abstraction, 71–72
  data hierarchies, 68–71
  defined, 65–66
  functions, 72–77
  information hiding, 68
  macros, 77–88
  object-oriented programming, 86–88
  simple, 67–68
  types of, 66–67
Abstraction barriers, 88
  CLOS, 72
  establishing, 71
Access operations, 43
Accessors, 32, 41–42
acons, 33
adjust-array, 35
Administrative files, 106–113
Algorithms, program efficiency and, 138–139
and, 40
Anonymous functions, 73–74
append, 32
apply, 56–58
apropos, 119–120
apropos-list, 119–120
aref, 35
Arithmetic functions, 74–75
Arrays, 2, 34–35. *See also* Vectors
  adjustable, 34
  defined, 34
  displaced, 34
  simple, 35
assert, 129
Assignment operators, 42–43
assoc, 32–33
Association lists
  for data storage, 33
  defined, 32–33
Attributes, 33
Auxiliary methods, 87

Backquote (`), 60
Backquote notation, 60–61, 77
Basic class, 69
Binary tree structure, 25
Bindings
  as abstraction, 66
  lexical, 43, 44
  variable, 43–44
bit, 35
Bit-vectors, 35
Body commentary, 104–105
break, 127
break-on-warnings, 128
Breakpoints, 127, 135
Bugs, 103
  commenting, 103
  reporting, 136

car, 32, 41, 42
case, 38, 39
ccase, 131

cdr, 32, 41, 42
cerror, 128–129
char-equal, 58–59
char=, 58–59
check-type, 129–130
Class definitions, 22–24
Class objects, 22
Class precedence list (CPL), 87
Classes, 35–37. *See also* Subclasses; Superclasses
  adding, 22–23
  basic, 69
  interned, 91–93
  mixin, 69
  object-oriented programming and, 87–88
  properties of, 87
  subclasses, 70
  superclasses, 70
CLOS, 10, 20
  abstraction in, 72
  abstraction barriers, 72
  classes, 36–37
  declaration files and, 113
  generic functions, 74–76
  inheritance, 91
  object-oriented programming, 87–88
  objects, 66
Code file organization, 105–113
Code walkers, 83
Command interface, 11
Comments
  body commentary, 104–105
  importance of, 99
  in-line, 105
  overview, 102–103
  self-commenting code, 100–101
  useful, 102–105
Common Lisp, 2, 53
Common Lisp Object System. *See* CLOS
Compilation
  debugging and, 144
  declarations and, 142
  eval-when and, 146–147
Compilation and load files, 107–110
compilation-speed, 143–144
compile, 146
cond, 38, 39, 40
Conditional expressions, 38–40
  defined, 38
  nested, 39
Conditions, 131
Cons, 41–42
  defined, 32
cons, 32
Consequents, 38
Constants, 45
Control structures, macros for, 79–80
copy-list, 32
CPL, 87
ctypecase, 131

Data abstraction, 66, 71–72
Data hierarchies, 68–71
Data manipulation problems, 141–142
Data structures, 32–37
  arrays, 34–35
  association lists, 32–33
  classes, 35–37
  effectiveness of, 31
  hash tables, 34–35
  hierarchies, 68–71
  lists, 32–34
  objects and, 21
  for *Personal Planner*, 21–25
  program efficiency and, 139
  property lists, 33–34
  structures, 35–37
  vectors, 34–35
Databases
  in *Personal Planner*, 12
  querying, 17–20
  storing, 15–16, 24–25
Date-parsing utility, 25–30
Debugging, 114–137
  compilation and, 144
  graceful degradation, 126
  learning from, 136–137
  methodology, 133–137
  style and, 114
  tools, 115–132
Debugging methodology, 133–137
Debugging tools, 115–132
  interactive, 115–126
  programmatic, 115, 126–132
Declarations, 101, 112–113
  defined, 142
  ftype, 143
  inline, 143
  optimize, 143–144
  program efficiency and, 142–146
  the, 143
  type, 142
declare, 142
decode-universal-time, 28
defclass, 13–14, 37, 83, 91, 112
defconstant, 45, 112
defgeneric, 37, 87
Defining forms, 77–79, 80–81
defmethod, 37
defpackage, 111
defparameter, 43, 45–47, 112
defstruct, 36, 83, 112
defvar, 43, 46–47, 112

describe, 115–116, 117
Descriptive names, 100
Design iteration, 14–15
Destructive modification, 63
do, 49
do*, 49
documentation, 48, 101
Documentation strings, 100–101
dolist, 49, 50
dotimes, 49, 54
Dynamic scoping, 5, 47

ecase, 130–131
Edit history, 102–103
Efficiency, 138–149
  compilation and, 142–147
  compounded problems and, 140–141
  data manipulation problems and, 141–142
  declarations and, 142–147
  generalization and, 148–149
  program, 138
  programmer, 138
  programming, 138–142
Empty list (), 41
encode-universal-time, 27
endp, 41
Entities
  class definition for, 22–23
  print method and, 70–71
Entity database, 24
Entity superclass, 24
eq, 58
eql, 58
equal, 58
Equality predicates, 58–59
equalp, 58
equalp hash table, 24
error, 128
Error-checking, 126
Error-handling facility, 126, 131–132
etypecase, 75, 130–131
eval, 57–58
eval-when, 84, 146–147
Evaluation control, 146–147
Event database, 25
Events
  class definition for, 23
  defining types of, 14
  storing, 25
Experimental code, writing, 20–30
export, 111

File accessors, 59
File options, 103
File organization, 105–113
Fill pointers, 34–35
first, 42
flet, 29, 51–52
format, 29
Free Software Foundation, 2
ftype declaration, 143
funcall, 56–58
function, 60
Functions
  as abstractions, 72–77
  anonymous, 73–74
  as first-class objects, 2
  generic, 74–77
  local, 51–52
  vs. macros, 82–83
  macros for defining, 78
  subroutining, 73
  user-callable, 72
  writing, 72–73

Generalization, 148–149
Generic functions, 37, 74–77
  arguments for, 75–76
  CLOS, 75
  defined, 74
  methods invoked by, 87
  print method and, 76
get, 33
get-properties, 33
getf, 33
gethash, 35
Global parameters, 45–46
Global values, 45
Global variables, 46–48
  naming conventions for, 101
GNU Emacs, 2, 103
go, 48
Graceful degradation, 126

Handlers, 131
Hash tables, 24, 34–35, 52
Home package, 110–111

Idioms
  common, 61–64
  creating, 64
if, 38, 39, 63
Implementation plan, writing, 12–13
import, 111
In-line coding, macros for, 82
In-line comments, 105
in-package, 110–112
Indentation, 101
Information hiding, 68, 88
Inheritance, 2, 23–24, 91
  multiple, 37
inline declaration, 82, 143

inspect, 75, 116–119, 124
Instances, 36
  creating, 88
Interfaces
  command, 11
  intermodule, 88–89
  programmatic, 11, 17–19
  user, 12, 19–20, 95–97
Interned classes, 91–93
Iteration
  vs. recursion, 55
  simple, 48–51

&keyword, 89

labels, 51, 55
Lambda expressions, 52–53
  as anonymous functions, 73–74
  mapping and, 51
let, 43–44, 53–54, 79
let*, 43–44
Lexical bindings, 43, 44
Lexical scoping, 5, 44
Lisp objects. *See* Objects
list, 32
list*, 32
List accessors, 32, 41–42
list-reverse, 75
Lists, 32–34. *See also* Association lists; Property lists
  defined, 32
  mapping and, 50, 53–54
  for storage, 32
load, 146
Local functions, 51–52
  mutually recursive, 51
  self-recursive, 51
Local transparency, 44, 48, 52, 53
Logical operators, 40–41
loop, 48–49, 54, 140–141
Loop keywords, 49
Loops, program efficiency and, 140–141
Lucid, Inc., 7
Lucid Common Lisp, 7
Lucid Common Lisp Advanced User's Guide, 111

Macro expansion
  defined, 77
  evaluation control and, 146–147
Macro syntax, 60–61
macroexpand, 84–85
macrolet, 52
Macros, 51–52, 77–86
  as abstractions, 66
  appropriate uses for, 77–80
  backquote notation for, 77
  for control structures, 79–80
  for controlling execution environment, 79
  for defining forms, 77–79, 80–81
  for defining functions, 82–83
  for defining objects, 80–81
  functions vs., 82–83
  harmful side effects, 83, 84–86
  inadvertent name capture of local variables with, 83
  as metaprograms, 77
  misuse of, 82–83
  multiple evaluation of macro arguments, 83–84
  problems to avoid when writing, 83–86
  redefining, 81
MACSYMA, 2
make-array, 35
make-hash-table, 35
make-instance, 37, 88
make-list, 32
mapc, 50, 54
mapcan, 50
mapcar, 50, 51, 55
mapcon, 50
maphash, 52
mapl, 50
maplist, 50
Mapping, 50–51
  lambda expressions and, 52, 73
member, 32
Metaprograms, macros as, 77
Method combination, 87
  standard, 76
Methods
  object-oriented programming and, 87–88
  print, 75–77
Mixins, 69
Multiple inheritance, 37
multiple-value-bind, 28

Names
  as abstraction, 66, 68
  conventions for, 101
  descriptive, 100
Nested conditional expressions, 39
Normalized random numbers list
  C program for, 3–4
  Lisp program for, 4–5, 53–55
not, 40, 41
null, 41

Object type hierarchy, 68–71
Object-oriented programming, 86–88
Objects
  abstraction of, 65–66
  in CLOS, 86–88
  Lisp, 2

Objects *(continued)*
methods associated with, 87–88
Personal Planner, 13–14, 21
typed, 2, 142–143
open, 59
Open coding, 143
Optimization, 144–146
optimize declaration, 143
&optional, 56, 89
or, 40, 41
Overview comments, 102–103

Package files, 110–112
pairlis, 33
Parameters, 45–46
Personal Planner, 7–8
code for, 151–208
compilation and load files, 107–110
data abstraction in, 71–72
data structure hierarchy in, 68–70
data structures for, 21–25
fine-tuning, 90–98
implementation plan for, 12–13
object-oriented programming for, 87
package files in, 111–112
print methods for, 76–77
program utilities for, 25–30
sample session of, 209–215
storage facility for, 15–17
top-level function of, 12
plist. *See* Property list
pop, 32
Precedence, object-oriented programming and, 87
Predicates, naming conventions for, 101
Primary methods, defined, 87
Print method, generic functions and, 76
print-object, 75
prog, 48
progn, 63
Program efficiency, 138
Program extensibility, 22, 137
Program files, organization of, 99, 113, 139
Program maintenance, 137
Program utilities, 25–30
Programmatic debugging tools, 115, 126–132
Programmatic interactions, initial description of, 10–12
Programmatic interface, 11, 17–19
Programmer efficiency, 138
Programming
efficiently, 138–142
object-oriented, 86–88
Programming idioms, 60–64
Programming problems
describing, 9–10
subdividing, 15–20
understanding, 9–15
Programs, defined, 1
Property lists
in data hierarchy, 68–70
defined, 32–33
psetf, 43
psetq, 43
push, 32

quote, 60
Quote ('), 60

Recursion, 51–52
vs. iteration, 55
tail, 52, 54
REDUCE, 2
remf, 33
remhash, 35
remprop, 33
Repetitive code, abstraction and, 67–68
Resources, 61–63
rest, 42
Restarts, 131
return, 49
room, 140
rplaca, 43
rplacd, 43

safety, 143–144
sbit, 35
schar, 35
Scoping. *See* Dynamic scoping; Lexical scoping
Self-commenting code, 100–101
Semicolons
four, for denoting a file subheading, 104
one, for denoting an in-line comment, 105
three, for denoting a description of succeeding construct, 104
three, for marking file overview, 102–103
two, for denoting a description of succeeding expression, 104–105
setf, 35, 42–43, 47
setq, 42–43, 47
shadow, 111
shadowing-import, 111
Side effects, macros and, 83, 84–86
Simple arrays, 35
Slot options, 37
Slots, 36, 37
abstraction barriers and, 72
defined, 35
space, 144
special declaration, 142
Special variables, 48
speed, 143–144
Standard method combination, 76

step, 122–125
Storage
  data structures for, 32–37
  of *Personal Planner* database, 15–16
Streams, 59
string-equal, 58–59
string=, 58–59
string-position, 27
Strings, 35
Structures, 35–36
  and CLOS, 36
Style
  debugging and, 114
  importance of, 1–2, 5–6
  Lisp and, 2–5
Subclasses, 69, 86
Subroutining, 73
Superclasses, 24, 69
  creating, 91
  defined, 86
svref, 35
symbol-plist, 33
symbol-value, 58
Symbols, 2, 5
  abstraction and, 66
  naming, 100
  packages and, 110–111
  property lists for, 33

t test, 40
Tail recursion, 52, 54
terminal-io, 120
the declaration, 143
time, 140
trace, 120–122
trace-output, 120
Transparency, local, 44, 48, 52, 53
Type checking, 142–143
type declaration, 142–143, 144–146
type slot option, 22
Typed objects, 2, 142–143
Typed variables, 4, 5
Typographical conventions, 7

Universal Time format, date-parsing utility for, 25–30
unless, 38, 39, 41
untrace, 122, 124
unwind-protect, 79
Update operations, 43
User interface, 19–20
  class definitions and, 94–97
User-callable functions, 72

values, 63–64
Values, global, 45

Variable bindings, 43–44
Variables
  global, 46–48
  special, 48
  typed, 4, 5
  value of, 2
vector, 35
Vectors, 34–35. *See also* Arrays
  for storage, 32
Virtual storage, 16

warn, 127–128
when, 38, 39
with-open-file, 59

LISP STYLE & DESIGN explores the process of style in the development of Lisp programs. Style comprises efficient algorithms, good organization, appropriate abstractions, well-constructed function definitions, useful commentary, and effective debugging. Good design and style enhance programming efficiency because they make programs easier to understand, to debug, and to maintain.

A special feature of this book is the large programming example that the authors use throughout to illustrate how the process develops: organizing the approach, choosing constructs, using abstractions, structuring files, debugging code, and improving program efficiency. Lisp programmers, symbolic programmers or those intrigued by symbolic programming, as well as students of Lisp should consider this book an essential addition to their libraries.

Molly M. Miller is Manager of Technical Publications and Training for Lucid, Inc. She holds degrees in Computer Science, Mathematics, and English and has done graduate work in symbolic and heuristic computation at Stanford University.

Eric Benson is Principal Scientist at Lucid, Inc. He is a graduate of the University of Utah with a degree in mathematics and is a co-founder of Lucid.

Also from Digital Press

COMMON LISP: The Language, Second Edition

Guy L. Steele Jr. Hardbound, EY-C194E-DP-SS. Paperback, EY-C187E-DP-SS.

A Programmer's Guide to COMMON LISP

Deborah G. Tatar. Paperback, EY-6706E-DP-SS.

digital™

DIGITAL PRESS

12 Crosby Drive Bedford, MA 01730

Order number EY-C199E-DP
DP ISBN 1-55558-044-0
PH ISBN 0-13-538422-2